

# Compact Deterministic Self-Stabilizing Leader Election: The Exponential Advantage of Being Talkative\*

Lélia Blin<sup>†</sup>

Université d'Evry-Val d'Essonne, 91000 Evry, France.

UPMC Sorbonne Universités, France.

LIP6-CNRS UMR 7606, France.

lelia.blin@lip6.fr

Sébastien Tixeuil

UPMC Sorbonne Universités, France.

Institut Universitaire de France.

LIP6-CNRS UMR 7606.

sebastien.tixeuil@lip6.fr

## Abstract

This paper focuses on *compact* deterministic self-stabilizing solutions for the leader election problem. When the protocol is required to be *silent* (i.e., when communication content remains fixed from some point in time during any execution), there exists a lower bound of  $\Omega(\log n)$  bits of memory per node participating to the leader election (where  $n$  denotes the number of nodes in the system). This lower bound holds even in rings. We present a new deterministic (non-silent) self-stabilizing protocol for  $n$ -node rings that uses only  $O(\log \log n)$  memory bits per node, and stabilizes in  $O(n \log^2 n)$  rounds. Our protocol has several attractive features that make it suitable for practical purposes. First, the communication model fits with the model used by existing compilers for real networks. Second, the size of the ring (or any upper bound on this size) needs not to be known by any node. Third, the node identifiers can be of various sizes. Finally, no synchrony assumption, besides a weakly fair scheduler, is assumed. Therefore, our result shows that, perhaps surprisingly, trading silence for exponential improvement in term of memory space does not come at a high cost regarding stabilization time or minimal assumptions.

---

\*A preliminary version of this paper has appeared in [9, 10].

<sup>†</sup>Additional support from the ANR project IRIS.

# 1 Introduction

This paper is targeting the issue of designing efficient self-stabilization algorithm for the leader election problem. *Self-stabilization* [15, 16, 32] is a general paradigm to provide forward recovery capabilities to distributed systems and networks. Intuitively, a protocol is self-stabilizing if it is able to recover from any transient failure, without external intervention. *Leader election* is one of the fundamental building blocks of distributed computing, as it enables to distinguish a single node in the system, and thus to perform specific actions using that node. Leader election is especially important in the context of self-stabilization as many protocols for various problems assume that a single leader exists in the system, even when faults occur. Hence, a self-stabilizing leader election mechanism enables to run such protocols in networks where no leader is a priori given, by using simple composition techniques [16].

Most of the literature in self-stabilization is dedicated to improving efficiency after failures occur, including minimizing the stabilization time, i.e., the maximum amount of time one has to wait before recovering from a failure. While stabilization time is meaningful to evaluate the efficiency of an algorithm in the presence of failures, it does not necessarily capture the overhead of self-stabilization when there are no faults [1], or after stabilization. Another important criterion to evaluate this overhead is the *memory space* used by each node. This criterion is motivated by two practical reasons, that we detail below.

First, self-stabilizing protocols require that *some* communications carry on forever (in order to be able to detect distributed inconsistencies due to transient failures [7, 14]). Therefore, minimizing the memory space used by each node enables to minimize the amount of information that is exchanged between nodes. Indeed, protocols are typically written in the *state model*, where the state of each node is read by each of its neighbors. (The use of the state model is motivated by the fact that all existing stabilization-preserving compilers [4, 11, 12, 31] are precisely designed for this model).

Second, minimizing memory space enables to significantly reduce the cost of redundancy when mixing self-stabilization and replication, in order to increase the probability of masking or containing transient faults [22, 23]. For instance, duplicating every bit three times at each node permits to withstand one randomly flipped bit. More generally, decreasing the memory space allows the designer to duplicate this memory many times, in order to tolerate many random bit-flips.

A foundational result regarding memory space in the context of self-stabilization is due to Dolev *et al.* [17], which states that,  $n$ -node networks,  $\Omega(\log n)$  bits of memory are required for solving global tasks such as leader election. Importantly, this bound holds even for the ring. A key component of this lower bound is that the protocol is assumed to be *silent*. (Recall that a protocol is silent if each of its executions reaches a point in time beyond which the registers containing the information available at each node do *not* change). The lower bound can be extended to *non-silent* protocols, but only for protocols with restricted capabilities. For instance, it holds in *anonymous* (and uniform) unidirectional rings, even of prime size [8, 21]. As a matter of fact, most deterministic self-stabilizing leader election protocols [2, 3, 5, 13, 18] use at least  $\Omega(\log n)$  bits of memory per node. Indeed, either these protocols directly compare node identifiers (and thus communicate node identifiers to neighbors), or they compute some variant of a hop-count distance to the elected node (and this distance can be as large as  $\Omega(n)$  to be accurate).

A few previous work [26, 27, 30] managed to break the  $\Omega(\log n)$  bits lower bound for the memory space of self-stabilizing leader election algorithms. Nevertheless, the corresponding

algorithms exhibit shortcomings that hinder their relevance to practical applications. For instance, the algorithm by Mayer *et al.* [30], by Itkis and Levin [26], and by Awerbuch and Ostrovsky [6] use a constant number of bits per node only. However, these algorithms guarantee *probabilistic* self-stabilization only (in the Las Vegas sense). In particular, the stabilization time is only *expected* to be polynomial in the size of the network. Moreover, these algorithms are designed for a communication model that is more powerful than the classical state model used in this paper. (The state model is the model used in most available compilers for actual networks [4, 11, 12, 31]). More specifically, Mayer *et al.* [30] use the message passing model, and Awerbuch and Ostrovsky [6] use the link-register model, where communications between neighboring nodes are carried out through dedicated registers. Finally, Itkis and Levin [26] use the state model augmented with reciprocal pointer to neighbors. In this model, not only a node  $u$  is able to distinguish a particular neighbor  $v$  (which can be done using local labeling), but also this distinguished neighbor  $v$  is aware that it has been selected by  $u$ . Implementing this mutual interaction between neighbors typically requires distance-two coloring, link coloring, or two-hops communication. All these techniques are impacting the memory space requirement significantly [29]. It is also important to note that the communication models in [6, 26, 30] allow nodes to send different information to different neighbors, while this capability is beyond the power of the classical state model. The ability to send different messages to different neighbors is a strong assumption in the context of self-stabilization. It enables to construct a “path of information” that is consistent between nodes. This path is typically used to distribute the storage of information along a path, in order to reduce the information stored at each node. However, this assumption prevents the user from taking advantage of the existing compilers. So implementing the protocols in [6, 26, 30] to actual networks requires to rewrite all the codes from scratch.

To our knowledge, the only *deterministic* self-stabilizing leader election protocol using sub-logarithmic memory space in the classical state model is due to Itkis *et al.* [27]. Their elegant algorithm uses only a constant number of bits per node, and stabilizes in  $O(n^2)$  time in  $n$ -node rings. However, the algorithm relies on several restricting assumptions. First, the algorithm works properly only if the size of the ring is *prime*. Second, it assumes that, at any time, a *single* node is scheduled for execution, that is, it assumes a *central* scheduler [20]. Such a scheduler is far less practical than the classical *distributed* scheduler, which allows any set of processes to be scheduled concurrently for execution. Third, the algorithm in [27] assumes that the ring is *oriented*. That is, every node is supposed to possess a consistent notion of left and right. This orientation permits to mimic the behavior of reciprocal pointer to neighbors mentioned above. Extending the algorithm by Itkis *et al.* [27] to more practical settings, i.e., to non-oriented rings of arbitrary size, to the use of a distributed scheduler, etc, is not trivial if one wants to preserve a sub-logarithmic memory space at each node. For example, the existing transformers enabling to enhance protocols designed for the central scheduler in order to operate under the distributed scheduler require  $\Theta(\log n)$  memory at each node [20]. Similarly, self-stabilizing ring-orientation protocols exist, but those preserving sub-logarithmic memory space either works only in rings of odd size for deterministic guarantees [24], or just provide probabilistic guarantees [25]. Moreover, in both cases, the stabilization time is  $O(n^2)$ , which is quite large.

To summarize, all existing self-stabilizing leader election algorithm designed in a practical communication model, and for rings of arbitrary size, without a priori orientation, use  $\Omega(\log n)$  bits of memory per node. Breaking this bound, without introducing any kind of restriction on the settings, requires, beside being non-silent, a completely new approach.

## Our results

In this paper, we present a deterministic (non-silent) self-stabilizing leader election algorithm that operates under the distributed scheduler in non-anonymous undirected rings of arbitrary size. Our algorithm is non-silent to circumvent the lower bound  $\Omega(\log n)$  bits of memory per node in [17]. It uses only  $O(\log \log n)$  bits of memory per node, and stabilizes in  $O(n \log^2 n)$  time.

Unlike the algorithms in [6, 26, 30], our algorithm is deterministic, and designed to run under the classical state-sharing communication model (a.k.a. state model), which allows it to be implemented by using actual compilers [4, 11, 12, 31]. Unlike [27], the size of the ring is arbitrary, the ring is not assumed to be oriented, and the scheduler is distributed. Moreover the stabilization time of our algorithm is significantly smaller than the one in [27]. Similarly to [6, 26, 28, 30], our algorithm uses a technique to distribute the information among nearby nodes along a sub-path of the ring. However, our algorithm does not rely on powerful communication models such as the ones used in [6, 26, 30]. Those powerful communication models make easy the construction and management of such sub-paths. The use of the classical state-sharing model makes the construction and management of the sub-paths much more difficult.

Besides the use of a sub-logarithmic memory space, and beside a quasi-linear stabilization time, our algorithm possesses several attractive features. First, the size (or any upper bound for this size) need not to be known by any node. Second, the node identifiers (or identities) can be of various sizes (to model, e.g., Internet networks running different versions of IP). Third, no synchrony assumption besides weak fairness is assumed (a node that is continuously enabled for execution is eventually scheduled for execution).

At a high level, our algorithm is essentially based on two techniques. One consists in electing the leader by comparing the identities of the nodes, bitwise, which requires special care, especially when the node identities can be of various sizes. The second technique consists in maintaining and merging trees based on a parenthood relation, and verifying the absence of cycles in the 1-factor induced by this parenthood relation. This verification is performed using small memory space by grouping the nodes in hyper-nodes of appropriate size. Each hyper-node handles an integer encoding a distance to a root. The bits of this distance are distributed among the nodes of the hyper-nodes to preserve a small memory per node. Difficulties arise when one needs to perform arithmetic operations on these distributed bits, especially in the context where nodes are unaware of the size of the ring. The precise design of our algorithm requires overcoming many other difficulties due to the need of maintaining correct information in an environment subject to arbitrary faults.

In addition, we took care of designing our algorithm to be ready for implementation, i.e., we do not only describe a conceptual protocol, but also produce a *concrete* self-stabilizing leader election protocol. This article provides a high level description of our algorithm, a detailed description of the protocol, and a complete proof of correctness.<sup>1</sup> To sum up, our result shows that, perhaps surprisingly, trading silence for exponential improvement in term of memory space does not come at a high cost regarding stabilization time, neither it does regarding minimal assumptions about the communication framework.

---

<sup>1</sup>Moreover, the reader is invited to consult [www-npa.lip6.fr/~blin/Election/](http://www-npa.lip6.fr/~blin/Election/) where a video of the dynamic execution of the protocol is presented. This video is the result of a complete implementation of the protocol. The video execution is using a randomized distributed scheduler. The initial configuration is illegitimate, and the video displays the protocol operation towards a legitimate configuration.

## 2 Model and definitions

### 2.1 Program syntax and semantics

A distributed system consists of  $n$  processors that form a communication graph. The processors are represented by the nodes of this graph, and the edges represent pairs of processors that can communicate directly with each other. Such processors are said to be *neighbors*. This classical model is called *state-sharing communication model*. The *distance* between two processors is the length (i.e., number of edges) of the shortest path between them in the communication graph. Each processor contains variables, and rules. A variable ranges over a fixed domain of values. A rule is of the form

$$\langle label \rangle : \langle guard \rangle \longrightarrow \langle command \rangle.$$

A *guard* is a boolean predicate over processor variables. A *command* is a set of variable-assignments. A command of processor  $p$  can only update its own variables. On the other hand,  $p$  can read the variables of its neighbors. An assignment of values to all variables in the system is called a *configuration*. A rule whose guard is **true** in some system configuration is said to be *enabled* in this configuration. The rule is *disabled* otherwise. An atomic execution of a subset of enabled rules results in a transition of the system from one configuration to another. This transition is called a *step*. A *run* of a distributed system is a sequence of transitions.

### 2.2 Schedulers

A *scheduler*, also called *daemon*, is a restriction on the runs to be considered. The schedulers differ among them by different execution semantics, and by different fairness in the activation of the processors [20]. With respect to execution semantics, we consider the least restrictive scheduler, called the *distributed scheduler*. In the run of a distributed scheduler, a step can contain the execution of an arbitrary subset of enabled rules of correct processors. With respect to fairness, we use the least restrictive scheduler, called *weakly fair scheduler*. In every run of the weakly fair scheduler, a rule of a correct processor is executed infinitely often if it is enabled in all but finitely many configurations of the run. That is, the rule has to be executed only if it is continuously enabled. A *round* is the smallest portion of an execution where every process has the opportunity to execute at least one action.

### 2.3 Predicates and specifications

A predicate is a boolean function over network configurations. A configuration *conforms* to some predicate  $R$ , if  $R$  evaluates to **true** in this configuration. The configuration *violates* the predicate otherwise. Predicate  $R$  is *closed* in a certain protocol  $P$ , if every configuration of a run of  $P$  conforms to  $R$ , provided that the protocol starts from a configuration conforming to  $R$ . Note that if a protocol configuration conforms to  $R$ , and the configuration resulting from the execution of any step of  $P$  also conforms to  $R$ , then  $R$  is closed in  $P$ .

A *specification* for a processor  $p$  defines a set of configuration sequences. These sequences are formed by variables of some subset of processors in the system. This subset always includes  $p$  itself. A *problem specification*, or *problem* for short, defines specifications for each processor of the system. A problem specification in the presence of faults defines specifications for correct processors only. Program  $P$  *solves* problem  $S$  under a certain scheduler if every run of  $P$  satisfies the specifications defined by  $S$ . A closed predicate  $I$  is an *invariant* of program  $P$  with respect to problem  $S$  if every run of  $P$  that starts in a state conforming to  $I$  satisfies  $S$ . Given two

predicates  $l_1$  and  $l_2$  for program  $P$  with respect to problem  $S$ ,  $l_2$  is an *attractor* for  $l_1$  if every run that starts from a configuration that conforms to  $l_1$  contains a configuration that conforms to  $l_2$ . Such a relationship is denoted by

$$l_1 \triangleright l_2.$$

A program  $P$  is *self-stabilizing* [15] to specification  $S$  if every run of  $P$  that starts in an arbitrary configuration contains a configuration conforming to an invariant of  $P$  with respect to problem  $S$ . That is, this invariant is an attractor of predicate *true*.

## 2.4 Leader election specification

Consider a system of processors where each processor has a boolean variable *leader*. We use the classical definition of *leader election*, which specifies that, in every protocol run, there is a suffix where a single processor  $p$  has  $\text{leader}_p = \text{true}$ , and every other processor  $q \neq p$  satisfies  $\text{leader}_q = \text{false}$ .

## 3 A compact leader-election protocol for rings

In this section, we describe our self-stabilizing algorithm for leader election in arbitrary  $n$ -node rings. The algorithm is later proved to use  $O(\log \log n)$  bits of memory per node, and to stabilize in quasi-linear time, whenever the identities of the nodes are between 1 and  $n^c$ , for some  $c \geq 1$ . For the sake of simplicity, we assume that the identifiers are in  $[1, n]$ . Nevertheless, the algorithm works without assuming any particular range for the identifiers. We first provide a general overview of the algorithm, followed by a more detailed description in Section 3.2. All predicates and commands are postponed in section 3.3.

### 3.1 Overview of the algorithm

As many existing deterministic self-stabilizing leader election algorithms, our algorithm aims at electing the node with maximum identity among all nodes, and, simultaneously, at constructing a spanning tree rooted at the elected node. The main constraint imposed by our wish to use sub-logarithmic memory is that we cannot exchange or even locally use complete identifiers, as their size  $\Omega(\log n)$  bits does not fit in a sub-logarithmic size memory. As a matter of fact, we assume that every node can access the bits of its identifier, but only a constant number of them can be simultaneously stored and/or communicated to neighbors at any given time. Our algorithm makes sure that every node stores the current position of a particular bit of the identifier, referred to as a *bit-position* in the sequel.

#### 3.1.1 Selection of the leader

Our algorithm operates in phases. At each phase, each node that is a candidate leader  $v$  reveals some bit-position, different from the ones at the previous phases, to its neighbors. More precisely, let  $\text{id}_v$  be the identity of node  $v$ , and assume that  $\text{id}_v = \sum_{i=0}^k b_i 2^i$ . Let  $I = \{i \in \{0, \dots, k\}, b_i \neq 0\}$  be the set of all non-zero bit-positions in the binary representation of  $\text{id}_v$ . Let us rewrite  $I = \{p_1, \dots, p_j\}$  with  $0 \leq p_1 < p_2 < \dots < p_j \leq k$ . Then, during Phase  $i$ ,  $i = 1, \dots, j$ , node  $v$  reveals  $p_{j-i+1}$  to its neighbors, which potentially propagate it to their neighbors, and possibly to the whole network in subsequent phases. During Phase  $i$ , for  $j+1 \leq i \leq \lfloor \log n \rfloor + 1$ , node  $v$  either becomes passive (that is, stops acting as a candidate leader) or remains a candidate

leader. If, at the beginning of the execution of the algorithm, all nodes are *candidate* leaders, then during each phase, some candidate leaders are eliminated, until exactly one candidate leader remains, which becomes the actual leader. More precisely, let  $p_{max}(i)$  be the most significant bit-position revealed at Phase  $i$  among all nodes. Then, among all candidate leaders still competing for becoming leader, only those whose bit-position revealed at Phase  $i$  is equal to  $p_{max}(i)$  carry on the electing process. The other ones become passive.

If all identities are in  $[1, n]$ , then the communicated bit-positions are less than  $\lceil \log n \rceil$ , and thus can be represented with  $O(\log \log n)$  bits. The difficulty is to implement this simple “compact” leader election mechanism in a self-stabilizing manner. In particular, the nodes may not have same number of bits encoding their identifiers, the ring may not start from a configuration where every node is a candidate leader, and the distributed scheduler may lead nodes to operate at various paces.

An additional problem in self-stabilizing leader election is the potential presence of *impostor* leaders. If one can store the identity of the leader at each node, then detecting an impostor is easy. Under our memory constraints, nodes cannot store the identity of the leader, nor read entirely their own identifier. So, detecting impostor leaders becomes non trivial, notably when an impostor has an identity whose most significant bit is equal to the most significant bit of the leader. To overcome this problem, the selection of the leader must run perpetually, leading our algorithm to be non-silent.

### 3.1.2 Spanning tree construction

Our approach to make the above scheme self-stabilizing is to merge the leader election process with a tree construction process. Every candidate leader is the root of a tree. Whenever a candidate leader becomes passive, its tree is merged to another tree, until there remains only one tree. The main obstacle in self-stabilizing tree-construction is to handle an arbitrary initial configuration. This is particularly difficult if the initial configuration yields a cycle rather than a spanning forest. In this case, when the leader election subroutine, and the tree construction subroutine are conjointly used, the presence of the cycle implies that, while every node is expecting to point to a neighbor leading to a leader, there are no leaders in the network. Such a configuration is called *fake* leader. In order to break cycles that can be present in the initial configuration, we use an improved variant of the classical distance calculation [19]. In the classical approach, every node  $u$  maintains an integer variable  $d_u$  that stores the distance from  $u$  to the root of its tree. If  $v$  denotes the parent of  $u$ , then typically  $d_v = d_u - 1$ , and if  $d_v \geq d_u$ , then  $u$  deletes its pointer to  $v$ . If the topology of the network is a ring, then detecting the presence of an initial spanning cycle, instead of a spanning forest, may involve distance variables as large as  $n$ , inducing  $\Omega(\log n)$  bits of memory.

In order to use exponentially less memory, our algorithm uses the distance technique but modulo  $\log n$ . More specifically, each node  $v$  maintains three variables. The first variable is an integer denoted by  $d_v \in \{0, \dots, \lfloor \log n \rfloor\}$ , called the “distance” of node  $v$ . Only candidate leaders  $v$  can have  $d_v = 0$ . Each node  $v$  maintains  $d_v = 1 + (\min\{d_u, d_{u'}\} \bmod \lfloor \log n \rfloor)$  where  $u$  and  $u'$  are the neighbors of  $v$  in the ring. Note that nodes are not aware of  $n$ . Thus they do not actually use the value  $\lfloor \log n \rfloor$  as above, but a potentially erroneous estimation of it.

The second variable is  $p_v$ , denoting the parent of node  $v$ . This parent is its neighbor  $w$  such that  $d_v = 1 + (d_w \bmod \lfloor \log n \rfloor)$ . By itself, this technique is not sufficient to detect the presence of a cycle, because the number of nodes can be a multiple of  $\lfloor \log n \rfloor$ . Therefore, we also introduce the notion of *hyper-node*, defined as follows:

**Definition 1** A hyper-node  $X$  is a set  $\{x_1, x_2, \dots, x_{\lfloor \log n \rfloor}\}$  of consecutive nodes in the ring, such that  $d_{x_1} = 1, d_{x_2} = 2, \dots, d_{x_{\lfloor \log n \rfloor}} = \lfloor \log n \rfloor, p_{x_2} = x_1, p_{x_3} = x_2, \dots, p_{x_{\lfloor \log n \rfloor}} = x_{\lfloor \log n \rfloor - 1}$  and  $p_{x_1} \neq x_2$ .

The parent of a hyper-node  $X = \{x_1, x_2, \dots, x_{\lfloor \log n \rfloor}\}$  is a hyper-node  $Y = \{y_1, y_2, \dots, y_{\lfloor \log n \rfloor}\}$  such that  $p_{x_1} = y_{\lfloor \log n \rfloor}$ . By definition, there are at most  $\lceil n / \lfloor \log n \rfloor \rceil$  hyper-nodes. If  $n$  is not divisible by  $\lfloor \log n \rfloor$ , then some nodes can be elements of an incomplete hyper-node. There can be several incomplete hyper-nodes, but if the parent of a (complete) hyper-node is an incomplete hyper-node, then an error is detected. Incomplete hyper-nodes must be leaves: there cannot be incomplete hyper-nodes in a cycle.

The key to our protocol is that hyper-nodes can maintain larger distance information than simple nodes, by distributing the information among the nodes of a hyper-node. More precisely, we assume that each node  $v$  maintains a bit of information, stored in variable  $dB_v$ . Let  $X = \{x_1, x_2, \dots, x_{\lfloor \log n \rfloor}\}$  be a hyper-node, the set  $dB_X = \{dB_{x_1}, dB_{x_2}, \dots, dB_{x_{\lfloor \log n \rfloor}}\}$  can be considered as the binary representation of an integer on  $\lfloor \log n \rfloor$  bits, i.e., between 0 and  $2^{\lfloor \log n \rfloor} - 1$ . Now, it is possible to use the same distance approach as usual, but at the hyper-node level. Part of our protocol consists in comparing, for two hyper-nodes  $X$  and  $Y$ , the distance  $dB_X$  and the distance  $dB_Y$ . If  $Y$  is the parent of  $X$ , then the difference between  $dB_X$  and  $dB_Y$  must be one. Otherwise an inconsistency is detected regarding the current spanning forest. The fact that hyper-nodes include  $\lfloor \log n \rfloor$  nodes implies that dealing with distances between hyper-nodes is sufficient to detect the presence of a cycle spanning the  $n$ -node ring. This is because  $2^{\lfloor \log n \rfloor} \geq n / \log n$ . (Note that hyper-nodes with  $k$  nodes such that  $2^k \geq n/k$  would do the same).

In essence, the part of our algorithm dedicated to checking the absence of a spanning cycle generated by the parenthood relation boils down to comparing distances between hyper-nodes. Note that comparing distances between hyper-node involves communication at distance  $\Omega(\log n)$ . This is another reason why our algorithm is non-silent.

## 3.2 Detailed description

### 3.2.1 Notations and preliminaries

Let  $C_n = (V, E)$  be the  $n$ -node ring, where  $V$  is the set of nodes, and  $E$  the set of edges. A node  $v$  has access to a unique identifier, but can only access to this identifier one bit at a time, using the  $\text{Bit}(x, v)$  function, that returns the position of the  $x$ th most significant bit equal to 1 in  $\text{Id}_v$ . This position can be encoded with  $O(\log \log n)$  bits when identifiers are encoded using  $O(\log n)$  bits, as we assume they are. A node  $v$  has access to local port number associated to its adjacent edges. The variable parent of node  $v$ , denoted by  $p_v$ , is actually the port number of the edge connecting  $v$  to its parent. In case of  $n$ -node rings,  $p_v \in \{0, 1\}$  for every  $v$ . (We do not assume any consistency between the port numbers). In a legitimate configuration, the structure induced by the parenthood relation must be a tree. The presence of more than one tree, or of a cycle, correspond to illegitimate configurations. We denote by  $N_v$  the set of the neighbors of  $v$  in  $C_n$ , for any node  $v \in V$ .

The variable distance, denoted by  $d_v$  at node  $v$ , takes values in  $\{-1, 0, 1, \dots, \lfloor \log n \rfloor\}$ . We have  $d_v = -1$  if all the variables of  $v$  are reset. We have  $d_v = 0$  if the node  $v$  is a root of some tree induced by the parenthood relation. Such a root is also called candidate leader. Finally,  $d_v \in \{1, \dots, \lfloor \log n \rfloor\}$  if  $v$  is a node of some tree induced by the parenthood relation, different from the root. Such a node is also called passive. Note that we only assume that variable  $d$  can hold



$\mathbb{R}_{\text{Error}} : \text{T.Er}(v) \vee \text{T.Reset}(v)$	$\rightarrow \text{Reset}(v);$
$\mathbb{R}_{\text{Start}} : \neg \text{T.Er}(v) \wedge \neg \text{T.Reset}(v) \wedge (d_v = -1) \wedge \text{T.Start}(v)$	$\rightarrow \text{Start}(v);$
$\mathbb{R}_{\text{Passive}} : \neg \text{T.Er}(v) \wedge \neg \text{T.Reset}(v) \wedge (d_v > -1) \wedge \text{T.Pass}(v)$	$\rightarrow \text{Passive}(v);$
$\mathbb{R}_{\text{Root}} : \neg \text{T.Er}(v) \wedge \neg \text{T.Reset}(v) \wedge (d_v = 0) \wedge \text{T.Pass}(v) \wedge \text{T.StartdB}(v)$	$\rightarrow \text{StartdB}(v);$
$\neg \text{T.Er}(v) \wedge \neg \text{T.Reset}(v) \wedge (d_v = 0) \wedge \neg \text{T.Pass}(v) \wedge \text{T.Inc}(v)$	$\rightarrow \text{Inc}(v);$
$\mathbb{R}_{\text{Update}} : \neg \text{T.Er}(v) \wedge \neg \text{T.Reset}(v) \wedge (d_v > 0) \wedge \neg \text{T.Pass}(v) \wedge \neg \text{EqElecP}(v) \wedge \text{T.Update}(v)$	$\rightarrow \text{Update}(v);$
$\mathbb{R}_{\text{HyperNd}} : \neg \text{T.Er}(v) \wedge \neg \text{T.Reset}(v) \wedge (d_v > 0) \wedge \neg \text{T.Pass}(v) \wedge \text{EqElecP}(v) \wedge$ $(\text{Add}_v = \emptyset) \wedge \text{T.Add}(v)$	$\rightarrow \text{BinAdd}(v);$
$(\text{Add}_v \neq \emptyset) \wedge \text{T.Broad}(v)$	$\rightarrow \text{Broad}(v);$
$\text{T.Verif}(v)$	$\rightarrow \text{Verif}(v);$
$\text{T.CleanM}(v)$	$\rightarrow \text{CleanM}(v);$

Figure 1: Formal description of algorithm **CLE**.

at least (and not exactly)  $\lfloor \log n \rfloor + 1$  different values, since nodes are not aware of how many they are in the ring, and just use an estimation of  $n$ . The children of a node  $v$  are returned by the macro  $\text{Ch}(v)$ , which returns the port number(s) of the edges leading to the child(ren) of  $v$ .

To detect cycles, we use four variables. First, each node maintains the variable  $\text{dB}$  introduced in the previous section, for constructing a distributed integer stored on an hyper-node. The second variable,  $\text{Add}_v \in \{+, \text{ok}, \emptyset\}$ , is used for performing additions involving values stored distributively on hyper-nodes. The third variable,  $\text{PL}_v$  (for pipeline), is used to send the result of an addition to the hyper-node children of the hyper-node containing  $v$ . Finally, the fourth variable,  $\text{HC}_v$  (for Hyper-node Checking), is dedicated to checking the hyper-node bits. Variables  $\text{PL}_v$  and  $\text{HC}_v$  are either empty, or each composed of a pair of variables  $(x, y) \in \{1, \dots, \lfloor \log n \rfloor\} \times \{0, 1\}$ .

For constructing the tree rooted at the node with highest identity, we use three additional variables. After convergence, we expect the leader to be the unique node with distance zero, and to be the root of an inward directed spanning tree of the ring, where the arc of the tree is defined by the parenthood relation. To satisfy the leader election specifications, we introduce the variable  $\text{leader}_v \in \{0, 1\}$  whose value is 1 if  $v$  is the leader and 0 otherwise. Since we do not assume that the identifiers of every node are encoded on the same number of bits, simply comparing the  $i$ -th most significant bit of two nodes is irrelevant. So, we use variable  $\hat{\text{B}}$ , which represents the most significant bit-position of all the identities present in the ring. This variable is also locally used at each node  $v$  as an estimate of  $\lfloor \log n \rfloor$ . Only the nodes  $v$  whose variable  $\hat{\text{B}}_v$  is equal to the most significant bit of the  $\text{Id}_v$  carry on participating to the election. Finally, the variables **Bit-Strong**, **Phase**, **Bit-Position** and **Control** are the core of the election process. Let  $r$  be the root of the tree including node  $v$ . Then, the variable  $\text{Bit-Strong}_v$  stores the position of the most significant bit of  $\text{Id}_r$ , variable  $\text{Phase}_v$  stores the current phase number  $i$ , variable **Bit-Position** stores the bit-position of  $\text{Id}_r$  at phase  $i$ , and variable  $\text{Control}_v$  stores a boolean dedicated to the control of the updating of the elections variables.

### 3.2.2 The Compact Leader Election algorithm **CLE**

Algorithm **CLE** is presented in Figure 1. In this figure, a rule of the form

$$\text{label} : \text{guard}_0 \wedge (\text{guard}_1 \vee \text{guard}_2) \longrightarrow (\text{command}_1; \text{command}_2)$$

where  $\text{command}_i$  is performed when  $\text{guard}_0 \wedge \text{guard}_i$  is true. Such a rule is presented in several lines, one for the common guard,  $\text{guard}_0$ , and one for each alternative guards,  $\text{guard}_i$ , with their respective command. Figure 1 describes the rules of the algorithm.

**CLE** is composed of six rules:

- The rule  $\mathbb{R}_{\text{Error}}$ , detects at node  $v$  the presence of inconsistencies between the content of its variables and the content of its neighboring variables. If  $v$  has not reset its variables, or has not restarted, the command  $\text{Reset}(v)$  is activated, i.e., all the content of all the variables at node  $v$  are reset, and the variable  $\mathbf{d}_v$  is set to  $-1$ .
- The rule  $\mathbb{R}_{\text{Start}}$ , makes sure that, if an inconsistency is detected at some node, then all the nodes of the network reset their variables, and restart. Before restarting, every node  $v$  waits until all its neighbors are reset or have restarted. A node  $v$  that restarts sets  $\mathbf{d}_v = 0$ , and its election variables  $\text{Bit-Strong}_v$ ,  $\text{Bit-Position}_v$  appropriately, with  $\text{Phase}_v = 1$ .
- The rule  $\mathbb{R}_{\text{Passive}}$ , is dedicated to the election process. A node  $v$  uses command  $\text{Passive}(v)$  when one of its neighbors has a bit-position larger than its bit-position, at the same phase.
- The rule  $\mathbb{R}_{\text{Root}}$ , concerns the candidate leaders, i.e., every node  $v$  with  $\mathbf{d}_v = 0$ . Such a node can only execute the rule  $\mathbb{R}_{\text{Root}}$ , resulting in that node performing one of the following two commands. Command  $\text{StartdB}(v)$  results in  $v$  distributing the bit  $\mathbf{dB}$  to its neighboring hyper-nodes. Command  $\text{Inc}(v)$  results in node  $v$  increasing its phase by 1. This happens when all the nodes between  $v$  and others candidate leaders in the current tree are in the same phase, with the same election values  $\text{Bit-Strong}_v$ ,  $\text{Bit-Position}_v$ ,  $\text{Control}_v$ .
- The rule  $\mathbb{R}_{\text{Update}}$ , is dedicated to updating the election variables.
- The rule  $\mathbb{R}_{\text{HyperNd}}$ , is dedicated to the hyper-nodes distance verification.

### 3.2.3 Hyper-nodes distance verification

Let us consider two hyper-nodes  $X$  and  $Y$  with  $X$  the parent of  $Y$ . Our technique for verifying the distance between the two hyper-nodes  $X$  and  $Y$ , is the following (see an example on Figure 2).  $X$  initiates the verification. For this purpose,  $X$  dedicates two local variables at each of its nodes: **Add** (to perform the addition) and **PL** (to broadcast the result of this addition inside  $X$ ). Similarly,  $Y$  uses the variable **HC** for receiving the result of the addition.

The binary addition starts at the node holding the last bit of  $X$ , that is node  $x_k$  with  $k = \widehat{\mathbf{B}}_v$ . Node  $x_k$  sets  $\text{Add}_{x_k} := +$ . Then, every node in  $X$ , but  $x_k$ , proceeds as follows. For  $k' < k$ , if the child  $x_{k''}$  of  $x_{k'}$  has  $\text{Add}_{x_{k''}} = +$  and  $\mathbf{dB}_{x_{k''}} = 1$ , then  $x_{k'}$  assigns  $+$  to  $\text{Add}_{x_{k'}}$ . Otherwise, if  $\text{Add}_{x_{k''}} = +$  and  $\mathbf{dB}_{x_{k''}} = 0$ , the binary addition at this point does not generate a carry, and thus  $x_{k'}$  assigns “ok” to  $\text{Add}_{x_{k'}}$ . Since  $\text{Add}_{x_{k'}} = \text{ok}$ , the binary addition is considered finished, and  $x_{k'}$ ’s ancestors (parent, grand-parent, etc.) in the hyper-node assign “ok” to their variable **Add**. However, if the first bit of  $X$  (that is,  $\mathbf{dB}_{x_1}$ ) is equal to one, then the algorithm detects an error because the addition would yield to an overflow. The result of the hyper-node binary addition is the following: if a node  $x_k$  has  $\text{Add}_{x_k} = \text{ok}$ , then it means that node  $y_k$  holds the appropriate bit corresponding to the correct result of the addition if and only if  $\mathbf{dB}_{y_k} = \mathbf{dB}_{x_k}$ . Otherwise, if  $\text{Add}_{x_k} = +$ , then the bit at  $y_k$  is correct if and only if  $\mathbf{dB}_{y_k} = \overline{\mathbf{dB}_{x_k}}$ <sup>2</sup>.

The binary addition in  $X$  is completed when node  $x_1$  satisfies  $\text{Add}_{x_1} = +$  or  $\text{Add}_{x_1} = \text{ok}$ . In that case,  $x_1$  starts broadcasting the result of the addition. For this purpose, it sets  $\text{PL}_{x_1} = (1, \mathbf{dB}_{y_1})$  where  $\mathbf{dB}_{y_1}$  is obtained from  $\text{Add}_{x_1}$  and  $\mathbf{dB}_{x_1}$ . Each node  $x_i$  in  $X$ ,  $i > 1$ , then successively perform the same operation as  $x_1$ . While doing so, node  $x_i$  sets  $\text{Add}_{x_i} = \perp$ , in order to enable the next verification. When the child of a node  $x_i$  publishes  $(\mathbf{d}_{x_i}, \mathbf{dB}_{x_i})$ , node  $x_i$  deletes  $\text{PL}_{x_i}$ , in order to, again, enable the next verification. From  $i = 1, \dots, k$ , all variables  $\text{PL}_{x_i}$  in  $X$  are deleted. When  $y_i$  sets  $\text{HC}_{y_i}[0] = \mathbf{d}y_i$ , node  $y_i$  can check whether the bit in  $\text{HC}_{y_i}[1]$  corresponds to  $\mathbf{dB}_{y_i}$ . If yes, then the verification carries on. Otherwise  $y_i$  detects a fault.

---

<sup>2</sup>If  $\mathbf{dB}_x = 0$  then  $\overline{\mathbf{dB}_x} = 1$ , and if  $\mathbf{dB}_x = 1$  then  $\overline{\mathbf{dB}_x} = 0$ .

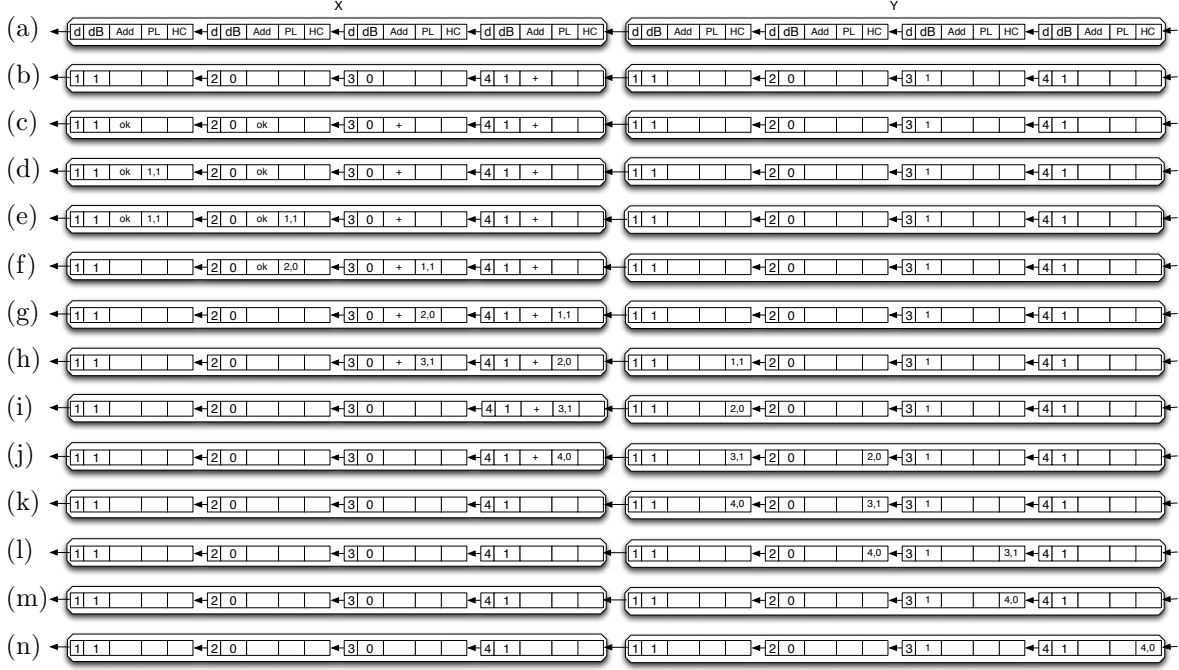


Figure 2: An example of distance verification between the hyper-node  $X$  and its child  $Y$ . In this example, hyper-nodes are composed of four nodes. (a) The memory of each node is represented by three boxes storing, respectively, the distance of the node to the root, the bit-distance of the hyper-node the node belongs to, the binary addition information, plus two boxes containing information for the bit verification. (Empty boxes contains  $\perp$ ). (b) The last node of  $X$  starts the addition. (c) The addition in  $X$  is completed. (d) The first node of  $X$  starts the verification. (e-g) The result of the addition is pipelined. (h) The first node  $v$  of  $Y$  checks  $\text{dB}_v$ . (j) The second node  $v'$  of  $Y$  checks  $\text{dB}_{v'}$ . (l) The third node  $v''$  of  $Y$  checks  $\text{dB}_{v''}$ . (n) The last node of  $Y$  detects an error.

### 3.2.4 Leader election and tree construction

As previously mentioned, our leader election protocol simultaneously performs, together with the election of a leader, the construction a tree rooted at the leader. The leader should be the node whose identifier is maximal among all nodes in the ring. Our assumptions regarding identifiers are very weak. In particular, identifiers may be of various sizes, and the total number  $n$  of different identifiers is not known to the nodes. In our algorithm, we use the variable  $\hat{B}$  to estimate (to some extent) the logarithm of the network size, and the variable  $\text{Bit-Strong}_v$  to propagate this estimation in the ring. More precisely,  $\hat{B}$  represents the most significant bit-position among all identities present in the ring, and we consider that all the nodes that do not carry the right value of  $\hat{B}$  in their local variables are not consistent. During the execution of the algorithm, only nodes whose identifiers match the most significant bit-position remain candidate leaders. Moreover, only candidate leaders broadcast bit-position during subsequent phases.

Let us now detail the usage of the variables used for the election. For the sake of simplification, we introduce the variable  $\text{Elec}$ . The variable  $\text{Elec}$  is equal to a 4-tuple:

$$\text{Elec}_v = (\text{Bit-Strong}_v, \text{Phase}_v, \text{Bit-Position}_v, \text{Control}_v)$$

This variable is essentially meant to represent the current bit-position of the candidate leaders.  $\text{Bit-Strong}_v$  represents the most significant bit-position among all identifiers, which must be in

agreement with variable  $\widehat{B}_v$  to assess the validity of  $\text{Elec}_v$ . The variables  $\text{Phase}_v$  and  $\text{Bit-Position}_v$  of  $\text{Elec}_v$  are the current phase  $i$ , and the corresponding bit-position revealed by a candidate leader during phase  $i$ , respectively. The comparison of bits-positions is relevant only if these bits-positions are revealed at the same phase. Hence, we force the system to proceed in phases. If, at phase  $i$ , the bit-position  $\rho_v$  of node  $v$  is smaller than the bit-position  $\rho_u$  of a neighboring node  $u$ , then node  $v$  becomes passive, and  $v$  takes  $u$  as parent. It is simple to compare two candidate leaders when these candidate leaders are neighbors. Yet, along with the execution of the algorithm, some nodes become passive, and therefore the remaining candidate leaders can be far away, separated by passive nodes. Each passive node is with a positive distance variable  $d$ , and is in a subtree rooted at some candidate leader. Let us now consider one such subtree  $T_v$  rooted at a candidate leader  $v$ . Whenever  $v$  increases its phase from  $i$  to  $i + 1$ , and sets the bit-position related to phase  $i + 1$ , all nodes  $u$  in  $T_v$  must update their variable  $\text{Elec}_u$  in order to have the same value as  $\text{Elec}_v$ .

At each phase, trees are merged into larger trees. At the end of phase  $i$ , all the nodes in a given tree have the same bit-position, and the leaves of the tree inform their parent that the phase is finished. The local variable **Control**, called control, is dedicated to this purpose. Each leaf assigns 1 to its control variable, and a bottom-up propagation of this control variable eventually reaches the root. In this way, the root learns that the current phase is finished. If the largest identifiers are encoded using  $\log n$  bits, each phase results in halving the number of trees, and therefore of candidate leaders. So within at most  $\log n$  phases, a single leader remains. To avoid electing an impostor leader, the (unique) leader restarts the election at the first phase. This is repeated forever. If an arbitrary initial configuration induces an impostor leader  $\ell$ , either  $\ell$  has not the most significant bit-position in its identifier or this impostor leader has its most significant bit-position equal to the most significant bit-position of the (real) leader. In the former case, the error is detected by a node with the most significant bit-position. In the latter case, error is detected by at least one node (the true leader), because there exists at least one phase  $i$  where the bit-position of the leader is superior to the bit-position of the impostor.

The process of leader election and spanning tree construction is slowed down by the hyper-node construction and management. When a node  $v$  changes its parents, it also changes its variable  $dB_v$ , in order not to impact the current construction of the tree. The point is that variable  $dB_v$  should be handled with extra care to remain coherent with the tree resulting from merging different trees. To handle this, every candidate leader assigns bits for its children into its variable **PL**. More precisely, if a root  $v$  has not children, then  $v$  publishes the bit for its future children with variable distance equal to one. If root  $v$  has children with distance variable equal to one, then  $v$  publishes the bit for the children  $u$  with  $d_u = 2$ , and so on, until the distance variable of  $v$  becomes  $\widehat{B}_v$ . On the other hand, a node cannot change its parent if its future new parent does not publish the bit corresponding to its future distance variable. When the hyper-node adjacent to the root is constructed, the hyper-node verification process takes care of the assignment of the bits to the node inside the hyper-node.

### 3.3 Formal description of our algorithm CLE

In the definitions below, the notation  $b \equiv P$  is used to define the boolean  $b$ , which is true if and only if the predicate  $P$  is true.

### 3.3.1 Computing the set of children:

$$\begin{aligned}
\text{Eq}\widehat{\mathbf{B}}(v) &= \{u \in \mathbf{N}_v \mid \widehat{\mathbf{B}}_u = \widehat{\mathbf{B}}_v \wedge \text{Bit-Strong}_u = \text{Bit-Strong}_v\} \\
\text{EqElec}(v, X) &= \{u \in X \mid \text{Elec}_u = \text{Elec}_v\} \\
\text{EqElecP}(v) &\equiv \text{Elec}_v = \text{Elec}_{p_v} \\
\text{InfPh}(v) &= \{u \in \text{Eq}\widehat{\mathbf{B}}(v) \mid (\text{Phase}_u = \text{Phase}_v - 1) \vee ((\text{Phase}_v = 1) \wedge (\text{Phase}_u = \widehat{\mathbf{B}}_v))\} \\
\text{CandC}(v) &= \text{EqElec}(v, \text{Eq}\widehat{\mathbf{B}}(v)) \cup \text{InfPh}(v) \\
\text{Ch}(v) &= \{u \in \text{CandC}(v) \mid ((d_v < \widehat{\mathbf{B}}_v) \wedge (d_u = d_v + 1)) \vee (d_v \in \{0, \widehat{\mathbf{B}}_v\} \wedge (d_u = 1))\}
\end{aligned}$$

### 3.3.2 Hyper-nodes distance verification

$$\begin{aligned}
\text{VCh}(v, M, T) &\equiv \{u \in \text{Ch}(v) \mid M = T\} = \text{Ch}(v) \\
\text{Add}_{\text{Nd}}(v) &\equiv (\text{Add}_v = \perp) \wedge (\text{PL}_v = \perp) \wedge (\text{Ch}(v) \neq \emptyset) \\
\text{Add}_p(v) &\equiv ((d_v > 1) \wedge (\text{Add}_{p_v} = \perp)) \vee (d_v = 1) \\
\text{Add}_{\text{Ch}}(v) &\equiv ((d_v < \widehat{\mathbf{B}}_v) \wedge \text{VCh}(v, \text{PL}, \perp)) \vee ((d_v = \widehat{\mathbf{B}}_v) \wedge \text{VCh}(v, \text{HC}, \perp)) \\
\text{Add}_+(v) &\equiv (d_v = \widehat{\mathbf{B}}_v) \vee (\text{VCh}(v, \text{Add}, +) \wedge \text{VCh}(v, \text{dB}, 1)) \\
\text{Add}_{ok}(v) &\equiv \text{VCh}(v, \text{Add}, ok) \vee (\text{VCh}(v, \text{Add}, +) \wedge \text{VCh}(v, \text{dB}, 0)) \\
\text{T.Add}(v) &\equiv \text{Add}_{\text{Nd}}(v) \wedge \text{Add}_p(v) \wedge \text{Add}_{\text{Ch}}(v) \wedge (\text{Add}_+(v) \vee \text{Add}_{ok}(v)) \\
\text{MCh}(v, k) &\equiv ((d_v < \widehat{\mathbf{B}}_v) \wedge \text{VCh}(v, \text{PL}, k)) \vee ((d_v = \widehat{\mathbf{B}}_v) \wedge \text{VCh}(v, \text{HC}, k)) \\
\text{Broad}_{\text{dB}}(v) &\equiv [(d_v = 1) \wedge (\text{PL}_v = \perp)] \vee [(d_v > 1) \wedge (\text{PL}_v[0] = d_v - 1) \wedge \text{MCh}(v, \text{PL}_v)] \\
\text{Broad}_{p_1}(v) &\equiv (\text{PL}_v = \perp) \wedge (\text{PL}_{p_v}[0] = 1) \wedge \text{MCh}(v, \emptyset) \\
\text{Broad}_{p_g}(v) &\equiv \text{PL}_v \wedge \text{MCh}(v, \text{PL}_v) \wedge (\text{PL}_{p_v}[0] = \text{PL}_v[0] + 1) \wedge (\text{PL}_v[0] \neq d_v - 1) \\
\text{Broad}_p(v) &\equiv (d_v > 1) \wedge (\text{Broad}_{p_1}(v) \vee \text{Broad}_{p_g}(v)) \\
\text{T.Broad}(v) &\equiv (\text{Ch}(v) \neq \emptyset) \wedge (\text{Broad}_{\text{dB}}(v) \vee \text{Broad}_p(v)) \\
\text{Vrf}_{\text{Last}}(v, M) &\equiv \neg \text{Ch}(v) \wedge (\text{Mp}_v[0] = d_v + 1) \\
\text{Vrf}_{\text{Start}}(v, M) &\equiv \text{Ch}(v) \wedge (\text{Mp}_v[0] = d_v + 1) \wedge \text{VCh}(v, \text{HC}, \emptyset) \\
\text{Vrf}_{\text{Broad}}(v, M) &\equiv \text{Ch}(v) \wedge (\text{Mp}_v[0] > d_v) \wedge \text{VCh}(v, \text{HC}, \text{HC}_v) \\
\text{Vrf}(v, M) &\equiv (\text{Vrf}_{\text{Last}}(v, M) \vee \text{Vrf}_{\text{Start}}(v, M) \vee \text{Vrf}_{\text{Broad}}(v, M)) \\
\text{Vrf}_1(v) &\equiv (d_v = 1) \wedge (\text{PL}_{p_v}[0] = \text{HC}_v[0] + 1) \wedge \text{Vrf}(v, \text{PL}) \\
\text{Vrf}_g(v) &\equiv (d_v > 1) \wedge (\text{HC}_{p_v}[0] = \text{HC}_v[0] + 1) \wedge (\text{Elec}_{p_v} = \text{Elec}_v) \wedge \text{Vrf}(v, \text{HC}) \\
\text{Vrf}_{1g}(v) &\equiv ((d_v = 1) \wedge (\text{PL}_{p_v}[0] = 1)) \vee ((d_v > 1) \wedge (\text{HC}_{p_v}[0] = d_v)) \\
\text{T.Verif}(v) &\equiv [(\text{HC}_v = \emptyset) \wedge \text{Vrf}_{1g}(v)] \vee [(\text{HC}_v \neq \emptyset) \wedge (\text{Vrf}_1(v) \vee \text{Vrf}_g(v))] \\
\text{EqElecN}(v, X) &\equiv \{u \in X \mid (\text{Elec}_u = \text{Elec}_v)\} = \mathbf{N}_v \\
\text{Clean}_{\text{M}_{v1A}}(v) &\equiv \text{EqElecN}(v, \text{Eq}\widehat{\mathbf{B}}(v)) \wedge (\text{HC}_v[0] = \widehat{\mathbf{B}}_v) \wedge [d_v = \widehat{\mathbf{B}}_v \vee (d_v \neq \widehat{\mathbf{B}}_v \wedge \text{VCh}(v, \text{HC}, (-1, -1)))] \\
\text{Clean}_{\text{M}_{v1B}}(v) &\equiv \text{EqElecN}(v, \text{Eq}\widehat{\mathbf{B}}(v)) \wedge \text{HC}_v = (\widehat{\mathbf{B}}_v, 0) \wedge [d_v = \widehat{\mathbf{B}}_v \vee (d_v \neq \widehat{\mathbf{B}}_v \wedge \text{VCh}(v, \text{HC}, \perp))] \\
\text{VHC}(v) &\equiv \text{HC}_v = (-1, -1) \wedge \text{HC}_{p_v} = (-1, -1) \\
\text{Clean}_{\text{M}_{vA}}(v) &\equiv \text{VHC}(v) \wedge [(d_v < \widehat{\mathbf{B}}_v \wedge \text{VCh}(v, \text{HC}, \emptyset)) \vee (d_v = \widehat{\mathbf{B}}_v \wedge \text{VCh}(v, \text{PL}, \perp))] \\
\text{Clean}_{\text{M}_{vB}}(v) &\equiv \text{HC}_v = (-1, -1) \wedge (d_v = 1) \wedge \text{VCh}(v, \text{HC}, \perp) \\
\text{Clean}_{\text{M}_{vD}}(v) &\equiv (d_v < \widehat{\mathbf{B}}_v) \wedge (\text{Ch}(v) \neq \emptyset) \wedge \text{VCh}(v, \text{HC}, \perp) \\
\text{Clean}_{\text{M}_{vC}}(v) &\equiv (\text{HC}[0] = \widehat{\mathbf{B}}_v) \wedge (((d_v = \widehat{\mathbf{B}}_v) \wedge (\text{Ch}(v) = \emptyset)) \vee \text{Clean}_{\text{M}_{vD}}(v)) \\
\text{Clean}_{\text{M}_C}(v) &\equiv (\text{PL}_v[0] = d_v) \wedge \text{MCh}(v, \text{PL}_v) \wedge (((\text{PL}_{p_v} = \perp) \wedge (d_v > 1)) \vee (d_v = 1)) \\
\text{T.CleanM}(v) &\equiv \text{Clean}_{\text{M}_{v1A}}(v) \vee \text{Clean}_{\text{M}_{v1B}}(v) \vee \text{Clean}_{\text{M}_{vA}}(v) \vee \text{Clean}_{\text{M}_{vB}}(v) \vee \text{Clean}_{\text{M}_C}(v) \\
\text{Root}(v) &\equiv (\text{leader}_v = 1) \wedge (d_v = 0) \wedge (p_v = \emptyset) \wedge (\widehat{\mathbf{B}}_v = \text{Bit}(1, \text{Id}_v)) \\
\text{T.StartdB}(v) &\equiv \text{Root}(v) \wedge (\text{Ch}(v) \neq \emptyset) \wedge \text{VCh}(v, \text{HC}, \text{PL}_v)
\end{aligned}$$

### 3.3.3 Leader election and tree construction

$\text{Max}\hat{B}(v)$	$= \max\{\text{Bit-Strong}_u \mid u \in N_v\}$
$\text{NeigMax}\hat{B}(v)$	$= \{u \in N_v \mid \text{Bit-Strong}_u = \text{Max}\hat{B}(v)\}$
$\text{EqPh}(v, X)$	$= \{u \in X \mid (\text{Phase}_u = \text{Phase}_v)\}$
$\text{MaxEqP}(v)$	$= \max\{\text{Bit-Position}_u \mid u \in \text{EqPh}(v, N_v) \wedge \text{Bit-Position}_u > \text{Bit-Position}_v\}$
$\text{NeigMaxEqP}(v)$	$= \{u \in \text{MaxEqP}(v) \mid \text{Bit-Position}_u = \text{MaxEqP}(v)\}$
$\text{Best}(v)$	$= \begin{cases} \min\{\text{port}_u \mid u \in \text{NeigMax}\hat{B}(v)\} & \text{if } \text{Max}\hat{B}(v) > \hat{B}_v \\ \min\{\text{port}_u \mid u \in \text{NeigMaxEqP}(v)\} & \text{otherwise} \end{cases}$
$\text{Pass}_0(v, x)$	$\equiv ((d_x = 0) \wedge (\text{PL}_x = (1, 0)))$
$\text{Pass}_{dB}(v, x)$	$\equiv \text{Pass}_0(v, x) \vee ((0 < d_x < \hat{B}_x) \wedge (\text{HC}_x[0] = d_x + 1)) \vee ((d_x = \hat{B}_x) \wedge (\text{PL}_x[0] = 1))$
$\text{EqMEi}(i, X)$	$= \{i \in \{0, \dots, i\} : \text{Elec}_u[i] = \text{Elec}_v[i] \mid u \in X\}$
$\text{EqMEx}(i, x, X)$	$= \{\text{Elec}_u[i] = x \mid u \in X\}$
$\text{Wave}_B(v)$	$\equiv (\text{EqMEi}(2, \text{Ch}(v)) = \text{Ch}(v) \wedge \text{EqMEx}(3, 1, \text{Ch}(v)) = \text{Ch}(v)) \vee \neg \text{Ch}(v)$
<b>T.Pass(v)</b>	<b><math>\equiv \text{Best}(v) \wedge \text{Pass}_{dB}(v, \text{Best}(v)) \wedge \text{Wave}_B(v)</math></b>
$\text{SupPh}(v)$	$= \{u \in \text{Eq}\hat{B}(v) \mid (\text{Phase}_u = \text{Phase}_v + 1) \vee (\text{Phase}_v = \hat{B}_v \wedge \text{Phase}_u = 1)\}$
$\text{Other}(v)$	$= \{u \in N_v - \text{Ch}(v) - \text{SupPh}(v) \mid \text{Elec}_v = \text{Elec}_u\}$
<b>T.Inc(v)</b>	<b><math>\equiv \text{Root}(v) \wedge \text{Wave}_B(v) \wedge (\text{Ch}(v) \cup \text{SupPh}(v) \cup \text{Other}(v) = N_v)</math></b>
$\text{Coh}_p(v)$	$\equiv (\text{Best}(v) = \emptyset) \wedge (p_v \in N_v) \wedge (\hat{B}_v = \hat{B}_{p_v}) \wedge (d_v = d_{p_v} - 1)$
$\text{Up}_p(v)$	$\equiv (\text{Control}_{p_v} = 0) \wedge ((\text{Phase}_{p_v} = \text{Phase}_v + 1) \vee ((\text{Phase}_v = \hat{B}_v) \wedge (\text{Phase}_{p_v} = 1)))$
$\text{Up}_E(v)$	$\equiv (\text{Control}_v = 0) \wedge (\text{Control}_{p_v} = 0)$
$\text{Up}_{\text{Back}}(v)$	$\equiv \text{Up}_E(v) \wedge (\text{EqMEi}(v, 2, N_v) = N_v) \wedge (\text{EqMEx}(2, \text{Ch}(v), 1) = \text{Ch}(v))$
<b>T.Update(v)</b>	<b><math>\equiv \text{Coh}_p(v) \wedge (\text{Up}_p(v) \wedge \text{Wave}_B(v)) \vee \text{Up}_{\text{Back}}(v)</math></b>

### 3.3.4 Reset and Error detection

$\text{MReset}(v)$	$\equiv (dB_v = 0) \wedge (\text{Add}_v = \perp) \wedge (\text{HC}_v = \perp)$
$\text{VReset}(v)$	$\equiv (\text{leader}_v = 0) \wedge (d_v = -1) \wedge (\hat{B}_v = -1) \wedge (\text{Elec}_v = (-1, 0, -1, -1))$
$\text{NdReset}(v)$	$\equiv \text{MReset}(v) \wedge (\text{PL}_v = \perp) \wedge \text{VReset}(v)$
$\text{NdStart}(v)$	$\equiv \text{MReset}(v) \wedge (\text{PL}_v \neq \perp) \wedge \text{Root}(v) \wedge (\text{Elec}_v = (\text{Bit}(1, \text{Id}_v), 1, \text{Bit}(1, \text{Id}_v), \{0, 1\}))$
$\text{NgReset}(v)$	$= \{u \in N_v \mid \text{NdReset}(u)\}$
$\text{NgStart}(v)$	$= \{u \in N_v \mid \text{NdStart}(u)\}$
<b>T.Reset(v)</b>	<b><math>\equiv \neg \text{NdReset}(v) \wedge \neg \text{NdStart}(v) \wedge  \text{NgReset}(v)  &gt; 0</math></b>
<b>T.Start(v)</b>	<b><math>\equiv \text{NdReset}(v) \wedge (\{\text{NgReset}(v) \cup \text{NgStart}(v)\} = N_v)</math></b>
$\text{PassNd}(v)$	$\equiv (\text{leader}_v = 0) \wedge (d_v > 0) \wedge \text{Coh}_p(v) \wedge (\hat{B}_v \geq \text{Bit}(1, \text{Id}_v)) \wedge (\text{Bit-Strong}_v = \hat{B}_v)$
$\text{Er}_d(v)$	$\equiv (d_v > 0) \wedge \wedge (\text{Best}(v) = \emptyset) \neg ((d_{p_v} < \hat{B}_v \wedge d_v \neq d_{p_v} + 1) \vee (d_{p_v} \in \{0, \hat{B}_v\} \wedge d_v \neq 1))$
$\text{Er}_{Nd}(v)$	$\equiv (\neg \text{Root}(v) \wedge \neg \text{PassNd}(v) \wedge \neg \text{NdReset}(v)) \vee (\text{PassNd}(v) \wedge \text{Er}_d(v))$
$\text{NgPh}(v)$	$= \{\text{Phase}_u \mid u \in N_v\}$
$\text{Er}_{\text{PhMinB}}(v)$	$\equiv ((\text{Phase}_v = \hat{B}_v) \wedge (\min\{\text{NgPh}(v)\} \notin \{1, \hat{B}_v - 1, \hat{B}_v\}))$
$\text{Er}_{\text{PhMinG}}(v)$	$\equiv ((\text{Phase}_v \neq \hat{B}_v) \wedge (\text{Phase}_v - \min\{\text{NgPh}(v)\} > 1))$
$\text{Er}_{\text{PhMax1}}(v)$	$\equiv ((\text{Phase}_v = 1) \wedge (\max\{\text{NgPh}(v)\} \notin \{2, \hat{B}_v\}))$
$\text{Er}_{\text{PhMaxG}}(v)$	$\equiv ((\text{Phase}_v \neq 1) \wedge (\max\{\text{NgPh}(v)\} - \text{Phase}_v > 1))$
$\text{Er}_{\text{Phase}}(v)$	$\equiv (\text{Er}_{\text{PhMinB}}(v) \vee \text{Er}_{\text{PhMinG}}(v) \vee \text{Er}_{\text{PhMax1}}(v) \vee \text{Er}_{\text{PhMaxG}}(v))$
$\text{Er}_{\text{Bp}}(v)$	$\equiv (d_v > 0) \wedge \{\{u \in \text{EqPh}(v, \text{Eq}\hat{B}(v)) \mid \text{Bit-Position}_u = \text{Bit-Position}_v\} \cup \text{SupPh}(v) \cup \{\text{Best}(v)\}\} = \emptyset$
$\text{Er}_{\text{Control}}(v)$	$\equiv ((\text{Control}_v = 1) \wedge \neg \text{Wave}_B(v)) \vee ((\text{Control}_v = 0) \wedge (\text{Control}_{p_v} = 1))$

$\text{Er}_{\text{MRoot}}(v)$	$\equiv (d_v = 0) \wedge ((\text{Add}_v \neq \perp) \vee (\text{HC}_v \neq \perp))$
$\text{Er}_{\text{MAdd}}(v)$	$\equiv (0 < d_v < \widehat{B}_v) \wedge (\text{Add}_v \neq \perp) \wedge \text{VCh}(v, \text{Add}, \perp)$
$\text{Er}_{\text{PL}}(v)$	$\equiv (d_v > 1) \wedge ((\text{PL}_v[0] > d_v) \vee ((\text{Best}(v) = \emptyset) \wedge (\text{PL}_{p_v} = \perp) \wedge (\text{PL}_v[0] < d_{p_v})))$
$\text{Er}_{\text{HCch}}(v)$	$\equiv (d_v < \widehat{B}_v) \wedge (\text{HC}_v = \perp) \wedge \text{VCh}(v, \text{HC}, \neg\perp)$
$\text{Er}_{\text{HCp}}(v)$	$\equiv (d_v > 0) \wedge (\text{HC}_v \neq \perp) \wedge (\text{Best}(v) = \emptyset) \wedge (\text{HC}_{p_v}[0] < \text{HC}_v[0])$
$\text{Er}_{\text{HC}}(v)$	$\equiv (0 < d_v < \widehat{B}_v) \wedge ((\text{HC}_v[0] < d_v) \vee \text{Er}_{\text{HCch}}(v) \vee \text{Er}_{\text{HCp}}(v))$
$\text{Er}_{\text{Mem}}(v)$	$\equiv \text{Er}_{\text{MRoot}}(v) \vee \text{Er}_{\text{MAdd}}(v) \vee \text{Er}_{\text{PL}}(v) \vee \text{Er}_{\text{HC}}(v)$
$\text{Er}_{\text{T}}(v)$	$\equiv (\text{NgReset}(v) = \emptyset) \wedge (\text{Er}_{\text{Nd}}(v) \vee \text{Er}_{\text{Phase}}(v) \vee \text{Er}_{\text{Bp}}(v) \vee \text{Er}_{\text{Control}}(v) \vee \text{Er}_{\text{Mem}}(v))$
$\text{Er}_{\text{Add}}(v)$	$\equiv (d_v = 1) \wedge (dB = 1) \wedge \text{VCh}(v, \text{Add}, +) \wedge \text{VCh}(v, dB, 1)$
$\text{Er}_{\text{H1}}(v)$	$\equiv (d_v = 1) \wedge (\text{PL}_{p_v}[0] = d_v) \wedge (\text{PL}_{p_v}[1] \neq dB_v)$
$\text{Er}_{\text{Hg}}(v)$	$\equiv (d_v > 1) \wedge (\text{HC}_{p_v}[0] = d_v) \wedge (\text{HC}_{p_v}[1] \neq dB_v)$
$\text{Er}_{\text{Hyper}}(v)$	$\equiv \text{Er}_{\text{Add}}(v) \vee ((\text{Elec}_v = \text{Elec}_{p_v}) \wedge (\text{Er}_{\text{H1}}(v) \vee \text{Er}_{\text{Hg}}(v)))$
$\text{Er}_{\text{Elec}}(v)$	$\equiv (d_v = 0) \wedge (\widehat{B}_v = \text{Bit}(1, \text{Id}_v)) \wedge (\text{Bit-Position}_v < \text{Bit}(\text{Phase}_v, \text{Id}_v))$
$\text{T.Er}(v)$	$\equiv \text{Er}_{\text{T}}(v) \vee \text{Er}_{\text{Hyper}}(v) \vee \text{Er}_{\text{Elec}}(v)$

### 3.3.5 Commands

#### Commands for hyper-node distance verification:

$\text{Verif}(v):$	$\text{BinAdd}(v):$
$(d_v = 1) \wedge (V_{\text{pd}}(v) \vee V_{\text{pi}}(v)) \rightarrow \text{HC}_v = \text{PL}_{p_v}$	$\text{Add}_+(v) \rightarrow \text{Add}_v = +$
$(d_v > 1) \wedge (V_{\text{pd}}(v) \vee V_{\text{pi}}(v)) \rightarrow \text{HC}_v = \text{HC}_{p_v}$	$\text{Add}_{ok}(v) \rightarrow \text{Add}_v = ok$
$\text{CleanM}(v):$	$\text{Broad}(v):$
$\text{CleanM}_{V1A}(v) \vee \text{CleanM}_{V1B}(v) \rightarrow \text{HC}_v = (-1, -1)$	$\text{Broad}_{dB}(v) \wedge (\text{Add}_v = ok) \rightarrow$
$\text{CleanM}_{VA}(v) \vee \text{CleanM}_{VB}(v) \rightarrow \text{HC}_v = \perp$	$\text{PL}_v = (d_v, dB_v); \text{Add}_v = \perp$
$\text{CleanM}_C(v) \rightarrow \text{PL}_v = \perp$	$\text{Broad}_{dB}(v) \wedge (\text{Add}_v = +) \rightarrow$
	$\text{PL}_v = (d_v, dB_v); \text{Add}_v = \perp$
	$\text{Broad}_p(v) \rightarrow \text{PL}_v = \text{PL}_{p_v}$

#### Commands for the leader election and tree construction:

$\text{Inc}(v):$	$\text{Passive}(v):$
$T \equiv (\text{Bit-Position}_v = -1) \wedge (\text{Phase}_v = \widehat{B}_v + 1)$	$\text{leader}_v := 0; p_v := \text{Best}(v); dB_v := \text{HC}_{p_v}[0];$
$T \rightarrow i := 1; \neg T \rightarrow i := \text{Phase}_v + 1;$	$(d_{p_v} = \widehat{B}_{p_v}) \rightarrow d_v := 1$
$\text{Elec}_v := (\text{Bit}(1, v), i, \text{Bit}(i, v), 0);$	$(d_{p_v} < \widehat{B}_{p_v}) \rightarrow d_v := d_{p_v} + 1$
$\text{StartdB}(v):$	$\widehat{B}_v := \widehat{B}_{p_v}; \text{Elec}_v := \text{Elec}_{p_v}$
$(\text{PL}_v[0] = \widehat{B}_v) \rightarrow \text{PL}_v := (1, 0)$	$\text{Add}_v := \perp; \text{HC}_v := \perp; \text{PL}_v := \perp;$
$(\text{PL}_v[0] < \widehat{B}_v) \rightarrow \text{PL}_v := (\text{PL}_v[0] + 1, 0)$	
$\text{Update}(v):$	
$\text{Wave}_B(v) \wedge \text{Up}_p(v) \rightarrow \text{Elec}_v = \text{Elec}_{p_v}$	
$\text{Up}_{\text{Back}}(v) \rightarrow \text{Control}_v = 1$	

#### Commands activated after error detection:

$\text{Reset}(v):$	$\text{Start}(v):$
$\text{leader}_v := 0; p_v := \emptyset; d_v := -1; dB_v := 0;$	$\text{leader}_v := 1; d_v := 0;$
$\widehat{B}_v := -1; \text{Elec}_v := (-1, 0, -1, -1);$	$\widehat{B}_v = \text{Bit}(1, v); \text{Elec}_v = (\text{Bit}(1, v), 1, \text{Bit}(1, v), 0);$
$\text{Add}_v := \perp; \text{HC}_v := \perp; \text{PL}_v := \perp;$	$\text{PL}_v = (1, 0);$

## 4 Correctness

In this section, we formally prove the correctness of our Algorithm.

**Theorem 1** *Algorithm **CLE** solves the leader election problem in a self-stabilizing manner for the  $n$ -node ring, in the state model, with a distributed weakly-fair scheduler. Moreover, if the  $n$  node identities are in the range  $[1, n^c]$  for some  $c \geq 1$ , then Algorithm **CLE** uses  $O(\log \log n)$  bits of memory per node, and stabilizes in  $O(n \log^2 n)$  rounds.*

The main difficulty for proving this theorem is to prove that **CLE** can detect any cycle generated by the parenthood relation in the initial configuration, and can, whenever a cycle is detected, remove this cycle. Let  $\Gamma$  be the set of all possible configurations of the ring, under the set of variables described before in the paper. First, we prove that Algorithm **CLE** detects the presence of “trivial” errors, that is, inconsistencies between neighbors. Second, we prove that, after correcting all the trivial errors (possibly using a reset), **CLE** converges and maintains configurations free of trivial errors. The set of configurations free of trivial errors is denoted by  $\Gamma_{\text{TEF}}$  where TEF stands for “Trivial Error Free”. From now on, we assume only configurations from  $\Gamma_{\text{TEF}}$ .

The core of the proof regarding proper cycle detection is based on proving the correctness of the hyper-node distance verification process. This verification process is the most technical part of the algorithm, and proving its correctness is actually the main challenge in the way of establishing Theorem 1. This is achieved by using proofs based on invariance arguments.

Once the correctness of the hyper-node distance verification process has been proved, we establish the convergence of Algorithm **CLE** from an arbitrary configuration in  $\Gamma_{\text{TEF}}$  to a configuration without cycles, and where all hyper-node distances are correct. The set of configurations without cycles is denoted by  $\Gamma_{\text{CF}}$  (where CF stands for “Cycle Free”). We prove that a configuration is in  $\Gamma_{\text{CF}}$  if and only if all hyper-node distances are correct. Once we can restrict ourselves to configurations in  $\Gamma_{\text{CF}}$ , we prove the correctness of our mechanisms detecting and removing impostor leaders. We denote by  $\Gamma_{\text{IEF}}$  (where IEF stands for “Impostor leader Error Free”) the set of configurations with no impostors. Finally, assuming a configuration in  $\Gamma_{\text{IEF}}$ , we prove that the system reaches and maintains a configuration with exactly one leader, equal to the node with maximum identity. Moreover, we prove that the structure induced by the parenthood relation is a tree rooted at the leader, and spanning all nodes. We denote by  $\Gamma_{\text{LE}}$  the set of configurations where the unique leader is the node with maximum identity. In other words, we prove that **CLE** is self-stabilizing for  $\Gamma_{\text{LE}}$ .

In the statements of the lemmas below, we define predicates on configurations, these predicates are used as attractors toward a legitimate configuration (i.e., a configuration with unique leader). To establish convergence toward attractors, we use potential functions [32], that is, functions that map configurations to non-negative integers, and that strictly decrease after each algorithm command is executed.

Before starting the proofs, let us first define the Predicate  $\Gamma_{\text{LE}}$  (*Leader Election*) that serves as the definition for legitimate configurations. Let  $L : \Gamma \rightarrow \mathbb{N}$  be the function defined by

$$L(\gamma) = \sum_{v \in V} \text{leader}_v.$$

A configuration  $\gamma$  is legitimate for the leader election specification, i.e., satisfies  $\Gamma_{\text{LE}}$ , if and only if  $L(\gamma) = 1$ . That is,

$$\Gamma_{\text{LE}} = \{\gamma \in \Gamma : L(\gamma) = 1\}.$$



Now, for the purpose of the proof, let us define Predicate  $\Gamma_{\text{TEF}}$  (*Trivial Error Free*). Let  $\psi : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\psi(\gamma, v) = \begin{cases} 1 & \text{if } \text{Er}_T(v) \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

Let  $\Psi : \Gamma \rightarrow \mathbb{N}$  be the function defined by:

$$\Psi(\gamma) = \sum_{v \in V} \psi(\gamma, v) .$$

Note that all nodes are legitimate with respect to  $\text{Er}_T(v)$  if and only if  $\Psi(\gamma) = 0$ . We define

$$\Gamma_{\text{TEF}} = \{\gamma \in \Gamma : \Psi(\gamma) = 0\}.$$

**Lemma 1** *true  $\triangleright \Gamma_{\text{TEF}}$  in one round.*

**Proof.** The predicate  $\text{Er}_T(v)$  is composed of two types of sub-predicates: the ones independent of the local variables of the neighbors, called *independent* sub-predicates, and the ones based on comparisons with the local variables of the neighbors, called *dependent* sub-predicates. (See predicate in 3.3.4). The independent sub-predicates are **Root**, **PassNd**, and **Er<sub>MRoot</sub>**. All other sub-predicates of  $\text{Er}_T$  are dependents. If an independent sub-predicate is true, then  $\text{Er}_T(v)$  is true, and the execution of any command by a neighbor of  $v$  has no influence on the predicate  $\text{Er}_T(v)$ , which remains true. (Note that if an independent sub-predicate of  $v$  is true, then  $v$  may have all its neighbors  $w$  with  $\text{Er}_T(w)$  equal to false). Let us now consider dependent sub-predicates. Let  $v$  a node with a dependent sub-predicate  $sp(v) = \text{true}$ . Then  $\text{Er}_T(v) = \text{true}$ , and there is at least one neighbor  $u$  of  $v$  such that some local variable(s) of  $u$  is not consistent with some variable(s) of  $v$ . In this case, since the sub-predicate  $sp$  is dependent, we also have  $sp(u) = \text{true}$ , as, if one variable of  $v$  is not consistent with the corresponding variable of  $u$ , then the converse is also true. Therefore,  $sp(u)$  is true, and thus  $\text{Er}_T(u)$  is true as well. As a consequence, nodes  $u$  and  $v$  can only apply the command **Reset**. Now let us consider the neighbor  $w$  of  $v$  distinct from  $u$ , and the neighbor  $y$  of  $u$  distinct from  $v$ . The predicates  $\text{Er}_T(w)$  and  $\text{Er}_T(y)$  may be equal to false, in which case  $w$  and  $y$  can execute a command different from **Reset**. Nevertheless, the execution of one command by node  $w$  or  $y$ , or both, cannot modify the variables of  $v$  and  $u$ . Therefore, the inconsistency between the local variables of  $v$  and  $u$  remains until one of them apply the reset command.

Let us denote by  $X$  the set of nodes  $v$  such that  $\text{Er}_T(v) = \text{true}$  (see predicate in 3.3.4) and  $\text{Er}_T(y) = \text{false}$  for every  $y \in \mathbf{N}_v$ . Moreover, let us denote by  $Y$  the set of nodes  $v$  such that  $\text{Er}_T(v) = \text{true}$  and there exists  $y \in \mathbf{N}_v$  such that  $\text{Er}_T(y) = \text{true}$ . Let  $\gamma_0$  denote the initial configuration of the system. Since the scheduler is weakly fair, all nodes in  $X$  are scheduled for execution at round 1. Moreover all nodes in  $Y$  are also scheduled at round 1. According to rule  $\mathbb{R}_{\text{Error}}$  (see algorithm in Figure 1), every node  $v \in X \cup Y$  executes **Reset**( $v$ ), and thus  $\text{Er}_T(v)$  becomes false for all these nodes as well as for all their neighbors (indeed, if one neighbor of  $v$  is reset, then  $\text{Er}_T(v)$  is false). Therefore, the configuration  $\gamma_1$  at time 1 satisfies

$$\Psi(\gamma_1) = 0.$$

Thus, starting from any arbitrary initial configuration  $\gamma_0$ , the system reaches a configuration  $\gamma_1 \in \Gamma_{\text{TEF}}$  in just one round.  $\square$

**Lemma 2**  $\Gamma_{\text{TEF}}$  is closed.

**Proof.** We prove that, starting from a configuration where  $\Gamma_{\text{TEF}}$  holds, algorithm **CLE** preserves that  $\text{Er}_{\text{Nd}}(v)$ ,  $\text{Er}_{\text{Phase}}$ ,  $\text{Er}_{\text{Bp}}$ ,  $\text{Er}_{\text{Control}}$ , and  $\text{Er}_{\text{Mem}}$  all remain false. Let us consider these predicates one by one, starting with  $\text{Er}_{\text{Nd}}(v)$ . We show that, for all  $v \in V$ , algorithm **CLE** keeps predicate  $\text{Er}_{\text{Nd}}(v)$  to false starting from a configuration where  $\Gamma_{\text{TEF}}$ . In configuration  $\Gamma_{\text{TEF}}$ , each node  $v$  has  $\text{Er}_{\text{Nd}}(v) = \text{false}$ , that is, each node  $v$  must satisfy one of the three predicates  $\text{Root}(v)$ ,  $\text{PassNd}(v)$ , and  $\text{maxElec}(v)$ . Moreover, if  $v$  satisfies  $\text{PassNd}(v) = \text{true}$ , then  $v$  must have a correct distance (see the definition of  $\text{Er}_{\text{d}}(v)$ ). We consider the different commands that can be executed by the algorithm.

- If a node  $v$  executes the command  $\text{Reset}$  due to a hyper-node distance error, i.e.,  $\text{Er}_{\text{Hyper}}(v) = \text{true}$ , or to an election error, i.e.,  $\text{Er}_{\text{Elec}}(v) = \text{true}$ , then the predicate  $\text{maxElec}(v)$  becomes true, and, as a consequence,  $\text{Er}_{\text{Nd}}(v)$  remains false. Let node  $u$  be a neighbor of  $v$ . Predicate  $\text{Er}_{\text{T}}(u)$  remains false because  $u$  has at least one neighbor that applied the reset command.
- A reset node  $v$ , i.e., a node  $v$  such that  $\text{d}_v = -1$  and  $\text{maxElec}(v) = \text{true}$ , can only execute the command  $\text{Start}(v)$ . When this command has been executed, the predicate  $\text{Root}(v)$  becomes true, and  $\text{d}_v$  becomes null.
- For the command  $\text{Inc}$  and  $\text{StartdB}$ , we note that the predicate  $\text{Root}(v)$  remains true if  $\text{d}_v = 0$  since no rule modifies the variables used in  $\text{Root}(v)$ .
- A candidate node  $v$ , i.e., a node  $v$  such that  $\text{d}_v = 0$  and  $\text{Root}(v) = \text{true}$ , can only execute the command  $\text{Passive}(v)$ . This command is activated if and only if  $v$  has a “better” neighbor  $u = \text{Best}(v)$ . In this case,  $v$  adjusts  $\text{d}_v$  with respect to  $\text{d}_u$ . As a consequence, we get  $\text{Er}_{\text{d}}(v) = \text{false}$ . Moreover, when the command  $\text{Passive}(v)$  has been executed, the predicate  $\text{Root}(v)$  becomes false, and the predicate  $\text{PassNd}(v)$  becomes true. Therefore,  $\text{Er}_{\text{Nd}}(v)$  remains false.
- The predicate  $\text{PassNd}(v)$  remains true because  $v$  executes the commands corresponding to  $\mathbb{R}_{\text{Passive}}$ ,  $\mathbb{R}_{\text{Update}}$ , or  $\mathbb{R}_{\text{HyperNd}}$ . We examine each of these different commands.
  - The commands corresponding to the rule  $\mathbb{R}_{\text{HyperNd}}$  do not change the variables used by  $\text{PassNd}(v)$  and  $\text{Er}_{\text{d}}(v)$ . Hence  $\text{Er}_{\text{Nd}}(v)$  remains false.
  - Note that for any two node  $u$  and  $v$  where  $u$  is a child of  $v$ , node  $u$  can only execute  $\text{Update}(u)$  after  $v$  has executed  $\text{Update}(v)$ . The command  $\text{Update}(v)$  keeps the predicate  $\text{PassNd}(v) = \text{true}$  because this command does not change the variables  $\text{leader}_v$ ,  $\text{p}_v$ ,  $\text{d}_v$ , and  $\widehat{\text{B}}_v$ . Moreover, a node  $v$  updates its variables according to the predicate  $\text{T.Update}(v)$  that satisfies  $\widehat{\text{B}}_v = \widehat{\text{B}}_{\text{p}_v}$ . So, after  $\text{Update}(v)$  has been performed, we have  $\text{Bit-Strong}_v = \widehat{\text{B}}_v$ . The command  $\text{Update}(v)$  does not modify the parent  $w$  of node  $v$ . Let  $u$  be the neighbor of  $v$  different from  $w$ . If  $u$  and  $w$  do not execute the command  $\text{Passive}$ , then  $\text{Best}(v) = \emptyset$  and  $\text{Er}_{\text{d}}(v)$  remains false. Note that the variable  $\text{Control}_v$  is not used in predicates  $\text{PassNd}(v)$  and  $\text{Er}_{\text{d}}(v)$ . If the parent  $w$  of  $v$  executes the command  $\text{Update}(w)$ , then  $\text{d}_w$  does not change, and, as a consequence,  $\text{Er}_{\text{d}}(v)$  remains false.
  - Let  $u$  be the neighbor of  $v$  such that  $u = \text{Best}(v)$ . If  $v$  executes  $\text{Passive}(v)$ , then  $\text{d}_v = \text{d}_u + 1$ . Therefore, the new distance of  $v$  is correct by definition of  $\text{Er}_{\text{d}}$ . Moreover since  $u = \text{Best}(v)$ , we have  $\widehat{\text{B}}_u \geq \text{Bit}(1, \text{Id}_v)$ . If the parent  $u$  of  $v$  executes  $\text{Passive}(u)$  and updates  $\text{d}_v$ , then  $u = \text{Best}(v)$ , and thus  $\text{Best}(v) \neq \emptyset$ . As a consequence,  $\text{PassNd}(v)$  remains true and  $\text{Er}_{\text{d}}$  remains false.

We now move on to predicate  $\text{Er}_{\text{Phase}}$ . We show that, for every  $v \in V$ , algorithm **CLE** keeps predicate  $\text{Er}_{\text{Phase}}(v) = \text{false}$ . An error is detected if and only if the difference between the phases of two nodes is larger than one. As before, we consider the different commands that can be executed, one by one, according to the value of  $\mathbf{d}_v$ .

- Let  $v$  be a node with  $\mathbf{d}_v = -1$  and  $\text{Phase}_v = 0$ . In this case, node  $v$  can only execute the command  $\text{Start}$ , which sets  $v$ 's phase to zero. In accordance with the rule  $\mathbb{R}_{\text{Start}}$ , each neighbor  $u$  of  $v$  has either reset (i.e.,  $\text{Phase}_u = 0$ ) or restarted (i.e.,  $\text{Phase}_u = 1$ ), which keeps  $\text{Er}_{\text{Phase}}(v) = \text{false}$ .
- Let  $v$  be a node with  $\mathbf{d}_v = 0$  and  $\text{Phase}_v = i$ , for some  $i \geq 0$ . In this case, node  $v$  can only execute the command  $\text{Passive}(v)$  or the command  $\text{Inc}(v)$ . In the latter case, the phase  $i$  of  $v$  can only be increased if  $\text{Phase}_u = i$  or  $i + 1$  for every neighbor  $u$  of  $v$ . Therefore, the command  $\text{Inc}(v)$  maintains  $\text{Er}_{\text{Phase}}(v) = \text{false}$ . We now consider command  $\text{Passive}(v)$ . When a  $v$  node chooses a neighbor  $u = \text{Best}(v)$  with  $\hat{\mathbf{B}}_u > \hat{\mathbf{B}}_v$ , the phase of  $u$  is equal to 1. Let us consider the subtree  $T_r$  rooted in  $r$ , and containing the node  $v$ , with possibly  $v = r$ . The node  $r$  increases its phase  $i$  if and only if all its descendants are in phase  $i$ , and the neighbors of the leaves of  $T_r$  are in phase  $i$  or  $i + 1$  (see the predicates  $\text{Wave}_{\mathbf{B}}$  and  $\text{Up}_{\text{Back}}$ ). As a consequence, the difference of phases between  $u = \text{Best}(v)$  and  $v$  is at most one. This keeps  $\text{Er}_{\text{Phase}}(v) = \text{false}$ . If  $\hat{\mathbf{B}}_u = \hat{\mathbf{B}}_v$ , then node  $u = \text{Best}(v)$  if its phase is the same as  $v$ , and its bit-position is greater than the bit-position of  $v$ . Therefore,  $\text{Er}_{\text{Phase}}(v)$  remains false.
- Let  $v$  be a passive node with  $\mathbf{d}_v > 0$  and  $\text{Phase}_v = i$ , for some  $i \geq 0$ . Only two commands can change the phase  $\text{Phase}_v$ : the command  $\text{Passive}(v)$ , and the command  $\text{Update}(v)$ . The command  $\text{Passive}(v)$  has already been considered in the previous item. Let us thus now consider the command  $\text{Update}(v)$ , and let  $r$  be the root of the subtree  $T_r$  containing  $v$ . When  $r$  increases its phase, all nodes in  $T_r$  have their phase equal to  $i$ . Let  $L_r$  be the set of leaves of  $T_r$ . All nodes  $u$  such that  $u \notin T_r$  with  $u \in \mathbf{N}_x$  for some  $x \in L_r$  have their phase equal either to  $i$  or to  $i + 1$  (this is ensured by the control variable  $\text{Control}_v$ ). As a consequence, if  $v$  moves to phase  $i + 1$ , then its parent is at phase  $i + 1$ . If  $v \notin L_r$ , then its children are at phase  $i$ . If  $v \in L_r$  then the neighbors of  $v$  not in  $T_r$  remain at phase  $i$  or at phase  $i + 1$ . Therefore,  $\text{Er}_{\text{Phase}}(v)$  remains false.

We now move on to predicate  $\text{Er}_{\text{Bp}}$ . We show that, for every  $v \in V$ , algorithm **CLE** keeps predicate  $\text{Er}_{\text{Bp}}(v) = \text{false}$ . The predicate  $\text{Er}_{\text{Bp}}(v)$  is satisfied if and only if  $\mathbf{d}_v > 0$ . A node  $v$  that is passive at phase  $i$  satisfies  $\text{Er}_{\text{Bp}}(v) = \text{false}$  if and only if  $v$  has not “better” neighbor (i.e.,  $\text{Best}(v) = \emptyset$ ), none of its neighbors are at phase  $i + 1$  (i.e.,  $\text{SupPh}(v) = \emptyset$ ), and at least one of its neighbors is at the same phase  $i$  with the same bit position as  $v$ . Note that, the commands corresponding to the rule  $\mathbb{R}_{\text{HyperNd}}$  do not change the variables used by the predicate  $\text{Er}_{\text{Bp}}(v)$ . As a consequence,  $\text{Er}_{\text{Bp}}(v)$  remains false. The only commands that may change  $\text{Er}_{\text{Bp}}$  are commands  $\text{Update}$ , and  $\text{Passive}$ . Let us again consider the subtree  $T_r$  rooted in  $r$  containing  $v$ , and let  $i$  be the current phase of the nodes in  $T_r$ . Note that, in this case,  $\text{Er}_{\text{Bp}}(v)$  is false because the parent of  $v$  has a same phase as  $v$ , and the same bit-position as  $v$ . If  $r$  increases its phase, then the ancestors of  $v$  update their phase top-down. Hence, when the parent  $u$  of  $v$  updates its phase, the predicate  $\text{SupPh}(v)$  becomes not empty. If  $r$  becomes passive, then ancestors of  $v$  have a “better” neighbor, and thus they set their variables according to the variables of their “better” neighbors. When the parent  $u$  of  $v$  executes the command  $\text{Passive}(u)$ , the predicate  $\text{Best}(v)$  becomes not empty. Hence, in all cases,  $\text{Er}_{\text{Bp}}(v)$  remains false.

We now turn our attention to predicate  $\text{Er}_{\text{Control}}$ . We prove that, for every  $v \in V$ , algorithm **CLE** keeps predicate  $\text{Er}_{\text{Control}}(v)$  to false. The boolean  $\text{Control}_v$  can only be modified by the command *Update* because of predicate **T.Update**.  $\text{Control}_v$  is used to control the updates of a subtree. Let us consider  $T$  the subtree containing  $v$ , and let  $r$  be the root of  $T$ . The updating process starts at node  $r$ , and all the descendants of  $r$  set their variables  $\text{Control} = 0$  top-down. The process ends at the leaves of the subtree  $T$  that launch the acknowledgement  $\text{Control} = 1$  propagated bottom-up to the root. Hence, when a node  $v$  satisfies  $\text{Control}_v = 1$ , all its descendants also satisfy  $\text{Control} = 1$ , and, conversely, when a node  $v$  satisfies  $\text{Control}_v = 0$ , all its ancestors satisfy  $\text{Control} = 0$ . Therefore,  $\text{Er}_{\text{Control}}(v)$  is kept to false.

Finally, we consider the predicate  $\text{Er}_{\text{Mem}}$ , and prove that, for every  $v \in V$ , algorithm **CLE** keeps predicate  $\text{Er}_{\text{Mem}}$  to false. The predicate  $\text{Er}_{\text{Mem}}$  is satisfied if and only if  $d_v \geq 0$ . The commands *Inc*( $v$ ), *StartdB*( $v$ ), and *Update*( $v$ ) do not change the variables  $\text{Add}_v$  and  $\text{HC}_v$ . Thus  $\text{Er}_{\text{Mem}}(v)$  remains false. The command *Passive* is executed during the construction of the spanning tree, when the hyper-nodes are set. Note that a hyper-node does not start the binary addition before it is properly set. When a node  $v$  becomes passive with  $\text{Best}(v) = u$ , the variables  $\text{Add}_v$  and  $\text{HC}_v$  are set to  $\perp$ . As a consequence, the predicate  $\text{Er}_{\text{MAdd}}(v)$  is false. Assume that  $v$  joins a hyper-node or that  $v$  is the first node of a hyper-node. In both cases,  $v$  has no children. Thus,  $\text{Er}_{\text{PL}}(v)$  and  $\text{Er}_{\text{HC}}(v)$  remain false. A neighbor  $w$  of  $v$  with  $p_w = v$  has now  $\text{Best}(w) = v$ . As a consequence the three predicates  $\text{Er}_{\text{MAdd}}(w)$ ,  $\text{Er}_{\text{PL}}(w)$  and  $\text{Er}_{\text{HC}}(w)$  remain false. We complete the proof by analyzing all commands corresponding to the rule  $\mathbb{R}_{\text{HyperNd}}$ .

- Let us consider the predicate  $\text{Er}_{\text{MAdd}}(v)$  at some node  $v$ . In this predicate,  $\text{Add}_v$  is compared to  $\text{Add}_u$  for every child  $u$  of  $v$ . Only two commands change  $\text{Add}_v$ : *BinAdd* and *Broad*. The command *BinAdd* assigns to  $\text{Add}$  the values  $+$  or *ok* in a bottom-up manner, starting at all nodes  $v$  satisfying  $d_v = \widehat{B}_v$ . The command *Broad* sets  $\text{Add}$  to  $\perp$  in top-down manner, starting at all nodes  $v$  with  $d_v = 1$ . As a consequence,  $\text{Er}_{\text{MAdd}}(v)$  remains false.
- Let us consider the predicate  $\text{Er}_{\text{PL}}(v)$ . In this predicate,  $\text{PL}_v$  is compared to  $\text{PL}_{p_v}$ . Only two commands can change  $\text{PL}_v$ : *Broad* and *CleanM*. The command *Broad* sets  $\text{PL}$  in a top-down manner starting at every node  $v$  with  $d_v = 1$ . The command *CleanM* sets  $\text{PL}$  to  $\perp$  whenever a child  $u$  of a node  $v$  has  $\text{PL}_u = \text{PL}_v$ . Let us consider a hyper-node  $X = \{x_1, \dots, x_k\}$ . At the end of the binary addition, all the variables  $\text{PL}$  of  $X$  are equal to  $\perp$ . The command *Broad*( $x_1$ ) sets  $\text{PL}_{x_1}[0] = d_{x_1}$ , which keeps  $\text{Er}_{\text{PL}}(x_1)$  at false because  $d_{x_1} = 1$ . The predicate  $\text{Broad}_p$  in the command *Broad*( $x_2$ ) sets  $\text{PL}_{x_2} = \text{PL}_{x_1}$  that maintains  $\text{Er}_{\text{PL}}(x_2)$  false because  $\text{PL}_{x_2}[0] = d_{x_1}$ . The node  $x_1$  can execute the command *CleanM* that sets  $\text{PL}_{x_1} = \perp$ . In this case,  $\text{Er}_{\text{PL}}(x_2)$  remains false. The node  $x_3$  executes *Broad*( $x_3$ ) and sets  $\text{PL}_{x_3}[0] = d_{x_1}$ . Then,  $x_2$  sets  $\text{PL}_{x_2}[0] = d_{x_2}$  and  $\text{Er}_{\text{PL}}(x_2)$  remains false. This process is repeated in a top-down manner for all the nodes in  $X$ . Hence, the algorithm **CLE** keeps  $\text{Er}_{\text{PL}}(v) = \text{false}$  for every node  $v$ .
- Finally, let us consider the predicate  $\text{Er}_{\text{HC}}(v)$ . In this predicate,  $\text{HC}_v$  is compared to  $\text{HC}_{p_v}$  and  $\text{HC}_u$  for every child  $u$  of  $v$ . The hyper-node distance is checked in a top-down manner. For that purpose, two commands are used: *Verif* and *CleanM*. Let us consider a hyper-node  $X = \{x_1, \dots, x_k\}$ . At the beginning of the hyper-node distance verification, all the variables  $\text{HC}$  of  $X$  are equal to  $\perp$ . According to the predicate **T.Verif**, every node  $x_i$ ,  $1 < i \leq k$ , sets  $\text{HC}_{x_i}[0] = \text{HC}_{x_{i-1}}[0]$  if and only if either  $\text{HC}_{p_{x_i}}[0] = d_{x_i}$ , or  $\text{HC}_{p_{x_i}}[0] = \text{HC}_{x_i}[0] + 1$ . In both cases  $\text{Er}_{\text{HC}}(v)$  remains false.

□

We now define Predicate  $\Gamma_{\text{CF}}$  (for *Cycle Free*), which ensures that no cycles induced by the  $\mathbf{p}$  variable remain in the network. Let  $\lambda : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\lambda(\gamma, v) = |\mathbf{dB}_{P_X} - \mathbf{dB}_X - 1|$$

Where  $X$  is an hyper-node, and  $P_X$  is the hyper-node parent of  $X$  (recall that  $\mathbf{dB}_X$  is an integer whose binary representation is  $\mathbf{dB}_{x_1}, \dots, \mathbf{dB}_{x_k}$  where  $k = \widehat{\mathbf{B}}$ ). Let  $\Lambda : \Gamma \rightarrow \mathbb{N}$  be the function defined by:

$$\Lambda(\gamma) = \sum_{v \in V} \lambda(\gamma, v).$$

Let  $\phi : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\phi(\gamma, v) = \begin{cases} |\mathbf{dp}_v - \mathbf{d}_v - 1| & \text{if } \mathbf{dp}_v < \widehat{\mathbf{B}}_v \\ |\mathbf{d}_v - 1| & \text{if } \mathbf{dp}_v = \widehat{\mathbf{B}}_v \end{cases}$$

Let  $\Phi : \Gamma \rightarrow \mathbb{N}$  be the function defined by

$$\Phi(\gamma) = \sum_{v \in V} \phi(\gamma, v).$$

We define

$$\Gamma_{\text{CF}} = \{\gamma \in \Gamma : \Psi(\gamma) = \Phi(\gamma) = \Lambda(\gamma) = 0 \text{ and } L(\gamma) > 0\}.$$

**Lemma 3**  $\Gamma_{\text{TEF}} \supset \Gamma_{\text{CF}}$  in  $O(n \log n)$  rounds.

**Proof.** Let us consider an initial configuration  $\gamma_0 \in \Gamma_{\text{TEF}}$  such that the overlay structure induced by the parent variables  $\mathbf{p}_v$ ,  $v \in V$ , forms a cycle  $C$ . The cycle  $C$  necessarily contains all nodes, which implies that all nodes have non empty pointers parent. Moreover, since  $\gamma_0 \in \Gamma_{\text{TEF}}$ , we get that, for every nodes  $v$ ,  $\text{leader}_v = 0$ . Thus  $L(\gamma_0) = 0$ .

Let  $\xi : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\xi(\gamma, v) = |\mathbf{Max}\widehat{\mathbf{B}} - \widehat{\mathbf{B}}_v|$$

for every  $\gamma \in \Gamma$ , and every  $v \in V$ , where  $\mathbf{Max}\widehat{\mathbf{B}} = \max\{\widehat{\mathbf{B}}_v \mid v \in V\}$ . Let then  $\Xi : \Gamma \rightarrow \mathbb{N}$  be the function defined by,

$$\Xi(\gamma) = \sum_{v \in V} \xi(\gamma, v)$$

We define

$$\Gamma_{\Xi} = \{\gamma \in \Gamma : \Xi(\gamma) = 0\}.$$

*Claim:*  $\Xi(\gamma_0) = 0$ .

Let us assume, for the purpose of contradiction, that  $\Xi(\gamma_0) \neq 0$ . Let

$$X = \{v \in V \mid \widehat{\mathbf{B}}_v = \mathbf{Max}\widehat{\mathbf{B}}\}.$$

Since  $\Xi(\gamma_0) \neq 0$ , we have  $X \neq V$ . Moreover, since  $C$  contains all nodes, we have that  $\mathbf{p}_v \neq \emptyset$  for every  $v \in V$ , and there is at least one node  $x \in X$  such that  $\mathbf{p}_x \notin X$ . This node  $x$  satisfies  $\widehat{\mathbf{B}}_x \neq \widehat{\mathbf{B}}_{\mathbf{p}_x}$ . In this case,  $\text{PassNd}(x) = \text{false}$  and  $\text{ErNd}(x) = \text{true}$ , which contradicts the fact that  $\gamma_0 \in \Gamma_{\text{TEF}}$ . Therefore,  $\Xi(\gamma_0) = 0$ .

Now, let  $\pi : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\pi(\gamma, v) = |\text{Bit-Strong}_v - \widehat{\text{MaxB}}| + |\text{Phase}_v - \text{MaxPh}| + |\text{Bit-Position}_v - \text{maxBp}| + |\text{Control}_v - \text{maxC}|$$

where

$$\begin{aligned} \text{MaxPh} &= \max\{\text{Phase}_v, v \in V\} \\ \text{MaxPhNd} &= \{v \in V \mid \text{Phase}_v = \text{MaxPh}\} \\ \text{MaxBp} &= \max\{\text{Bit-Position}_v, v \in \text{MaxPhNd}\} \\ \text{MaxBpNd} &= \{v \in \text{MaxPhNd} \mid \text{Bit-Position}_v = \text{MaxBp}\} \\ \text{MaxC} &= \max\{\text{Control}_v, v \in \text{MaxPhNd}\}. \end{aligned}$$

Beside, let us define the quadruplet

$$\text{MaxElec} = (\widehat{\text{MaxB}}, \text{MaxPh}, \text{MaxBp}, \text{MaxC}),$$

and the function  $\Pi : \Gamma \rightarrow \mathbb{N}$  such that

$$\Pi(\gamma) = \sum_{v \in V} \pi(\gamma, v).$$

We define

$$\Gamma_\Pi = \{\gamma \in \Gamma : \Pi(\gamma) = 0\}.$$

*Claim:*  $\Pi(\gamma_0) = 0$ .

Recall that  $\gamma_0 \in \Gamma_{\text{TEF}}$ ,  $\Xi(\gamma_0) = 0$ , and  $L(\gamma_0) = 0$ . Assume for the purpose of contradiction that  $\Pi(\gamma_0) \neq 0$ . Let

$$X = \{v \in V \mid \text{Elec}_v = \text{MaxElec}\}.$$

A direct consequence of  $\Pi(\gamma_0) \neq 0$  is that  $X \neq V$  and  $|X| < n$ . Since the cycle  $C$  contains all nodes, we have that, for every  $v \in V$ ,  $\mathbf{p}_v \neq \emptyset$ , and there is at least one node  $x \in X$  such that  $\mathbf{p}_x \notin X$ . We denote by  $y = \mathbf{p}_x$  the parent of  $x$ , and we consider the variables  $\text{Bit-Strong}_x$ ,  $\text{Phase}_x$ ,  $\text{Bit-Position}_x$ , and  $\text{Control}_x$ .

We have  $\text{Bit-Strong}_x = \widehat{\text{B}}_x$  because otherwise  $\text{PassNd}(v) = \text{false}$ , in contradiction with  $\gamma_0 \in \Gamma_{\text{TEF}}$ . Therefore  $\text{Bit-Strong}_x = \text{Bit-Strong}_y$  because  $\Xi(\gamma_0) = 0$ .

If  $|\text{Phase}_x - \text{Phase}_y| > 1$  then  $x$  or  $y$  detects an error (see the predicate  $\text{Er}_{\text{Phase}}(v)$ ), and thus  $\gamma_0 \notin \Gamma_{\text{TEF}}$ . If  $|\text{Phase}_x - \text{Phase}_y| = 1$ , then, since  $y \notin X$ , we have  $\text{Phase}_x = \text{Phase}_y + 1$ . Thus  $y$  cannot be a parent of  $x$  (see the predicate  $\text{Coh}_p$ ). Thus  $\gamma_0 \notin \Gamma_{\text{TEF}}$ . Therefore,  $\text{Phase}_x = \text{Phase}_y$ .

We now turn our attention to the variable  $\text{Bit-Position}_x$ . If  $|X| = 1$  then  $\text{Er}_{\text{Bp}}(x) = \text{true}$  because, on the one hand,  $x$  has no neighbors  $z$  such that  $\text{Phase}_z = \text{Phase}_x$  and  $\text{Bit-Position}_z = \text{Bit-Position}_x$ , and, on the other hand,  $x$  has no neighbors  $z$  such that  $\text{Phase}_z = \text{Phase}_x + 1$  and  $z = \text{Best}_x$  (see predicate  $\text{Best}$ ). Thus, if  $|X| = 1$  then  $\gamma_0 \notin \Gamma_{\text{TEF}}$ . Therefore,  $|X| > 1$ . Since  $\text{Phase}_x = \text{Phase}_y$  we have  $\text{Bit-Position}_x = \text{Bit-Position}_y$  because otherwise  $y$  cannot be a parent of  $x$  (see the predicate  $\text{Coh}_p$ ). Therefore  $\text{Bit-Position}_x = \text{Bit-Position}_y$ .

Finally, we consider  $\text{Control}_x$ , and show  $\text{Control}_x = \text{Control}_y$ . Assume that  $\text{Control}_x \neq \text{Control}_y$ . Then  $|X| > 1$  because otherwise Predicate  $\text{Er}_{\text{Control}}(x)$  would be true, in contradiction with  $\gamma_0 \in \Gamma_{\text{TEF}}$ . Moreover, if  $\text{Control}_y = 1$  and  $\text{Control}_x = 0$ , then  $\text{Er}_{\text{Control}}(x) = \text{true}$  and  $\text{Er}_{\text{Control}}(y) = \text{true}$ , again contradicting  $\gamma_0 \in \Gamma_{\text{TEF}}$ . If  $\text{Control}_y = 0$  and  $\text{Control}_x = 1$  then let  $x' \in X$  be a descendent of  $x$  whose child  $x'' \notin X$ . In this case,  $\text{Control}_{x'} = 1$  and  $\text{Control}_{x''} = 0$ ,

hence Predicate  $\text{Er}_{\text{Control}}(x') = \text{true}$ , in contradiction with  $\gamma \in \Gamma_{\text{TEF}}$ . Therefore  $\text{Control}_x = \text{Control}_y$ .

Since  $\text{Bit-Strong}_x = \text{Bit-Strong}_y$ ,  $\text{Phase}_x = \text{Phase}_y$ ,  $\text{Bit-Position}_x = \text{Bit-Position}_y$ , and  $\text{Control}_x = \text{Control}_y$ , we obtain that  $y \in X$ , in contradiction with  $p_x \notin X$ . As a consequence,  $\Pi(\gamma_0) = 0$ , which complete the proof of the claim.

An important consequence of  $\Pi(\gamma_0) = 0$  is that, for every node  $v$ , we have  $\text{Elec}_v = \text{MaxElec}$ .

*Claim:*  $\Phi(\gamma_0) = 0$ .

Again, the proof is by contradiction, assuming  $\Phi(\gamma_0) \neq 0$ . Since  $L(\gamma) = 0$ , there are no candidates for being the root of the tree. Thus Predicate  $\text{T.Inc}(v) = \text{false}$  for every node  $v$ , and therefore the command  $\text{Inc}(v)$  cannot be executed at any node  $v$ . As a consequence, all nodes are passive (i.e.,  $\forall v \in V : \text{PassNd}(v) = \text{true}$ ). In addition, since  $\text{Elec}_v = \text{MaxElec}$  for every  $v \in V$ , we also get that the commands  $\text{Passive}(v)$  and  $\text{Update}(v)$  cannot be executed at any node  $v$ . Since  $\Phi(\gamma_0) \neq 0$ , there exists at least one passive node  $v$  that detects an error between its distance and the distance of its parent (see predicate  $\text{Er}_d(v)$ ). Hence, for that node  $v$ , the predicates  $\text{PassNd}(v)$  and  $\text{Er}_{\text{Nd}}(v)$  are both true, which is a contradiction with  $\gamma_0 \in \Gamma_{\text{TEF}}$ . Thus  $\Phi(\gamma_0) = 0$ .

We are now ready to show that, if the initial configuration  $\gamma_0$  contains a cycle, then Algorithm **CLE** detects an error in  $O(n \log n)$  rounds.

Since  $\Phi(\gamma_0) = 0$ , we necessarily have that  $n$  is a multiple of  $\text{Max}\hat{\text{B}}$ , and that there are  $n/\text{Max}\hat{\text{B}}$  hyper-nodes. Since all nodes are passive in  $\gamma_0$ , the only commands that can be executed by some node(s) are related to the distance verification between hyper-nodes, that is Commands  $\text{BinAdd}(v)$ ,  $\text{Broad}(v)$ ,  $\text{Verif}(v)$  and  $\text{CleanM}(v)$ . More specifically, the only nodes that can possibly be activated in  $\gamma_0$  are the nodes  $v$  such that  $d_v = \hat{\text{B}}_v$ .

For every hyper-node  $X = (x_1, x_2, \dots, x_k)$ , where  $k = \text{Max}\hat{\text{B}}$ , since the scheduler is weakly fair, predicate  $\text{T.Add}(x_k) = \text{true}$ , and  $x_k$  executes the command  $\text{BinAdd}(x_k)$  at round 1. This yields the execution of the binary addition. The binary addition occurs from  $x_k$  to  $x_1$ , and every node in each hyper-node  $X$  eventually takes value "+" or "ok" once  $\text{Max}\hat{\text{B}}$  rounds has been performed. Now, if  $d_{x_1} = 1$  and  $\text{Add}_{x_1} = +$ , then an error is detected since the binary addition overflows beyond the limit of  $\text{Max}\hat{\text{B}}$  bits (see  $\text{Er}_{\text{MAdd}}(v)$ ).

Node  $x_1$  starts the verification process that propagates from  $x_1$  to  $x_k$ . Fix any hyper-node  $X = (x_1, x_2, \dots, x_k)$ , and let us denote by  $Y$  the hyper-node child of  $X$  in the current configuration at round  $\text{Max}\hat{\text{B}}$ . Node  $x_1$  computes the values of  $d_{y_1}$  (see the predicate  $\text{T.Broad}$ , and the command  $\text{Broad}$ ). This value is broadcast from  $x_1$  to  $x_k$  (see the predicate  $\text{T.Broad}$ , and the command  $\text{Broad}$ ). Node  $y_1$  checks whether  $\text{PL}_{x_k}[1] = d_{y_1}$ . If it is the case, then the verification process for all other nodes in  $Y$  carries on (see the predicate  $\text{T.Verif}$  and the command  $\text{Verif}$ ). Otherwise  $y_1$  detects an error (see  $\text{Er}_{\text{Hyper}}(v)$ ). Thanks to Predicates  $\text{T.Add}$ ,  $\text{T.Broad}$ , and  $\text{T.Verif}$ , and to commands  $\text{BinAdd}$ ,  $\text{Broad}$ , and  $\text{Verif}$  every node in  $Y$  are eventually checked, after an additional  $\text{Max}\hat{\text{B}}$  rounds. The total number of rounds for checking hyper-nodes is the following: there are  $n/\lceil \log n \rceil$  hyper-nodes, each hyper-node performs verification in  $O(\lceil \log n \rceil)$  rounds, so the overall process takes  $O(n \log n)$  rounds.

To sum-up, if the configuration  $\gamma_0$  contains a cycle, then at least one hyper-node detects an error in a  $O(n \log n)$  rounds.

Finally, if  $L(\gamma_0) = 0$  in an arbitrary configuration, then the algorithm **CLE** detects an error in  $O(n \log n)$  rounds, and, by Lemma 1, the system reaches a configuration in  $\Gamma_{\text{CF}}$ .  $\square$

**Lemma 4**  $\Gamma_{\text{CF}}$  is closed.

**Proof.** Let us consider a configuration  $\gamma \in \Gamma_{\text{CF}}$ . We already noticed in the proof of Lemma 2 that Algorithm **CLE** preserves coherent distances (i.e.,  $\Phi(\gamma) = 0$ ), and does not introduce trivial errors (i.e.,  $\Psi(\gamma) = 0$ ). Moreover, in the proof of Lemma 3, we have explained that the hyper-node distance verification correctly reports errors, if any. The variable  $\text{dB}$  is only modified by the command *Passive*. In the sequel, we use the wording “ $v$  joins  $T_r$ ” when a node  $v$  executes the command *Passive*, and the pointer  $\text{p}_v$  of  $v$  points to a node in the subtree  $T_r$  rooted at  $r$ .

We denote by  $X$  the set of candidate nodes. Let us first consider a node  $v$  (in  $X$  or not), and a root  $r \in X$  such that  $\text{d}_v \leq \lfloor \log n \rfloor$ , and  $v$  joins  $T_r$  when  $r$  increases its phase from  $i - 1$  to  $i$ , for some  $i$ . Thanks to Predicate **T.StartdB**, and to command *StartdB*, node  $r$  publishes first the bit for the node at distance 1 from  $r$ . In other words,  $\text{PL}_r = (1, 0)$ . When any node  $v$  at distance 1 joins  $T_r$ ,  $v$  sets  $\text{dB}_v = \text{PL}_r[1]$ , and then  $v$  informs  $r$  about the updating of  $\text{dB}$  by setting  $\text{PL}_v = \text{PL}_r$ . At this point,  $r$  can publish the bit for nodes at distance 2 (i.e.,  $\text{PL}_r = (2, 0)$ ), and so on until the distance reaches  $\lfloor \log n \rfloor$ . Now, a node  $v$  joins  $T_r$  only if its candidate parent publishes the bit that corresponds to the binary representation of the distance between  $v$  and  $r$ . In other words, for any node  $u$ , if  $\text{d}_u = k$  with  $k < \lfloor \log n \rfloor$  then **Bit-Strong** $_u$  must be equal to  $k + 1$ . This enables to maintain  $\Lambda(\gamma) = 0$ . When  $k$  reaches  $\lfloor \log n \rfloor$ , a hyper-node is created. Then, a binary addition process is carried out, and computes the bit for  $v$  when  $v$  joins  $T_r$ . This process maintains  $\Lambda(\gamma) = 0$ , and, as a direct consequence  $L(\gamma)$  remains greater than 0. To conclude, algorithm **CLE** keeps  $\Psi(\gamma) = 0$ ,  $\Phi(\gamma) = 0$ ,  $\Lambda(\gamma) = 0$ , and  $L(\gamma) > 0$ .  $\square$

We now introduce Predicate  $\Gamma_{\text{IEF}}$  (for *Impostor Error Free*), which ensures that the currently elected leader is not an impostor. Let  $\rho : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\rho(\gamma, v) = |\text{maxFB} - \widehat{\text{B}}_v|$$

where  $\text{maxFB} = \max\{\text{Bit}(1, \text{ld}_v) \mid v \in V\}$ . Let  $P : \Gamma \rightarrow \mathbb{N}$  be the function defined by

$$P(\gamma) = \sum_{v \in V} \rho(\gamma, v).$$

We show that **CLE** reaches a legitimate configuration with respect to leader election if and only if  $P(\gamma) = 0$ .

Let  $\epsilon : \Gamma \times V \rightarrow \mathbb{N}$  be the function defined by:

$$\epsilon(\gamma, v) = \begin{cases} 0 & \text{if } \text{d}_v = 0 \wedge \text{Bit}(\text{minPh}, \text{ld}_v) = \text{Bit}(\text{minPh}, l^*) \\ 1 & \text{if } \text{d}_v = 0 \wedge \text{Bit}(\text{minPh}, \text{ld}_v) < \text{Bit}(\text{minPh}, l^*) \\ 0 & \text{otherwise} \end{cases}$$

where  $\text{minPh} = \min\{\text{Phase}_v \mid v \in V\}$  and  $l^*$  is the identity of the node with the maximum identity. Let  $E : \Gamma \rightarrow \mathbb{N}$  be the function defined by,

$$E(\gamma) = \sum_{v \in V} \epsilon(\gamma, v).$$

We define

$$\Gamma_{\text{IEF}} = \{\gamma \in \Gamma : \Psi(\gamma) = P(\gamma) = E(\gamma) = 0\}.$$

**Lemma 5**  $\Gamma_{\text{CF}} \triangleright \Gamma_{\text{IEF}}$  in  $O(n \log n)$  rounds.



**Proof.** Let us consider first an initial configuration  $\gamma \in \Gamma_{\text{CF}}$ . We have observed in lemma 3 that being in  $\Gamma_{\text{CF}}$  implies  $L(\gamma) > 0$ . Let us denote by  $\ell^*$  the node with maximum identity, and  $X$  the set of the candidate nodes. Let us suppose that  $\ell^* \notin X$ . Let us denote by  $\ell$  the node with the maximum identity in  $X$ . In the worst case, all the sub-spanning tree merge in a unique spanning tree rooted at  $\ell$ . Thus, let us suppose that  $\gamma$  is a configuration where the network is spanned by an unique tree rooted in  $\ell$ . In this case  $d_\ell = 0$  and  $d_u > 0$  for every node  $u \neq \ell$ .

Let us assume, for the purpose of contradiction, that  $P(\gamma) \neq 0$ . If the tree is rooted at  $\ell$ , then every node must have the same  $\widehat{B}$  as  $\ell$ . Since  $\ell$  is a root,  $\widehat{B}_\ell = \text{Bit}(1, \text{ld}_\ell)$ . Hence,  $\widehat{B}_\ell \neq \text{maxFB}$  (because  $P(\gamma) \neq 0$ ). Now,  $\widehat{B}_\ell$  cannot be larger than  $\text{maxFB}$ , so there exists  $v$  such that  $\widehat{B}_v = \text{maxFB}$ , and  $\text{PassNd}(v)$  is true. This contradicts  $\gamma \in \Gamma_{\text{TEF}}$ , so we can conclude that  $P(\gamma) = 0$ .

Since  $\ell$  and  $\ell^*$  have the same number of bits, there must exist one phase where the bit-position of  $\ell^*$  is larger than the bit-position of  $\ell$ . More formally, there exists  $i$ ,  $1 < i \leq \lfloor \log n \rfloor + 1$ , such that, for every  $j < i$ , we have  $\text{Bit}(j, \text{ld}_\ell) = \text{Bit}(j, \text{ld}_{\ell^*})$ , and, for every  $k \geq i$ , we have  $\text{Bit}(k, \text{ld}_\ell) < \text{Bit}(k, \text{ld}_{\ell^*})$ . Note that  $i > 1$ , because, in  $\Gamma_{\text{TEF}}$ , predicate  $\text{PassNd}$  must be true. The worst case with respect to time complexity is for  $i = \lfloor \log n \rfloor + 1$ , and the arbitrary initial configuration starts at phase 2. In this case, only  $\mathbb{R}_{\text{Root}}$  can be activated for  $\ell$ , and only the rule  $\mathbb{R}_{\text{Node}}$  for the other nodes (in parallel to the hyper-node distance verification). Node  $\ell$  executes the command  $\text{Inc}(\ell)$  for  $\lfloor \log n \rfloor + 1 - 2$  times. After each execution of command  $\text{Inc}$ , every node executing this command updates the variable  $\text{Elec}$  in a top-down manner (see Predicate  $\text{T.Update}$  and command  $\text{Update}$ ). This updating process takes at most  $n$  rounds. When all nodes have the same election values, a bottom-up control process is initiated (see Predicate  $\text{T.Update}$  and command  $\text{Update}$ ). This process takes at most  $n$  rounds. After that,  $\ell$  increases its phase, and the same process is repeated. At the last phase,  $\text{Er}_{\text{Elec}}(\ell^*)$  is true, and an error is detected. Therefore, if the system contains a impostor leader, then an error is detected in  $O(n \log n)$  rounds.  $\square$

**Lemma 6**  $\Gamma_{\text{IEF}}$  is closed.

**Proof.** Let  $\gamma \in \Gamma_{\text{IEF}}$ , with  $L(\gamma) > 0$ . Let  $X$  be the set of candidate leaders (i.e., for every  $x \in X$ ,  $\text{Root}(x) = \text{true}$ ). Let  $x \in X$ , and let  $T_{(x,i)}$  be the subtree rooted at  $x$  during phase  $i$ . For a node  $v \in T_{(x,i)}$ , we have that (1) either  $\text{Bit-Strong}_v < \text{Bit-Strong}_x$  or  $\text{Bit-Strong}_v = \text{Bit-Strong}_x$ , and (2)  $\text{Bit}(j, \text{ld}_v) < \text{Bit}(j, \text{ld}_x)$  for every  $j \leq i$ . Moreover, for any two candidates leaders  $x_1$  and  $x_2$ , we have that, at phase  $i$ ,  $\text{Bit-Strong}_{x_1} = \text{Bit-Strong}_{x_2}$ , and  $\text{Bit}(j, \text{ld}_{x_1}) = \text{Bit}(j, \text{ld}_{x_2})$  for every  $j \leq i$ . Let  $i$  be such that  $T_{(x_1,i)}$  and  $T_{(x_2,i)}$  have adjacent nodes. Let  $x'_1 \in T_{(x_1,i)}$  and  $x'_2 \in T_{(x_2,i)}$  be two nodes such that  $x'_1$  is adjacent to  $x'_2$ . If, at phase  $i + 1$ ,  $\text{Bit}(i + 1, \text{ld}_{x_1}) > \text{Bit}(i + 1, \text{ld}_{x_2})$ , then the nodes of  $T_{(x_1,i)}$  and  $T_{(x_2,i)}$  activated by Predicate  $\text{T.Update}$  executes  $\text{Update}$ . When node  $x'_1$  reaches  $\text{Elec}_{x_1}$  at phase  $i + 1$ , node  $x'_2$  becomes passive (cf. command  $\text{Passive}$ ) and selects node  $x'_1$  as its parent. In a bottom-up fashion, every node of  $T_{(x_2,i)}$  joins the subtree  $T_{(x_1,i+1)}$  (cf. command  $\text{Passive}$ ), and, eventually,  $x_2$  becomes passive and joins  $T_{(x_1,i+1)}$ . By this process, for every node  $v$  in  $T_{(x_1,i+1)}$ , we have (1) either  $\text{Bit-Strong}_v < \text{Bit-Strong}_{x_1}$  or  $\text{Bit-Strong}_v = \text{Bit-Strong}_{x_1}$ , and (2)  $\text{Bit}(j, \text{ld}_v) < \text{Bit}(j, \text{ld}_{x_1})$  for every  $j \leq i + 1$ . This process is repeated until phase  $\lfloor \log n \rfloor$ , where there remains a single leader in the network.  $\square$

Note that if a node executes  $\text{Reset}$ , and since the scheduler is weakly fair, at most  $n$  rounds later all nodes have executed  $\text{Reset}$ .

**Lemma 7**  $\Gamma_{\text{IEF}} \triangleright \Gamma_{\text{LE}}$  in  $O(n \log^2 n)$  rounds.

**Proof.** We know by Lemma 4 that  $\Gamma_{\text{CF}}$  is closed, and we know by Lemma 6 that  $\Gamma_{\text{IEF}}$  is closed. Moreover the proof of Lemma 6 provided details about the election process at each phase. Let  $\gamma \in \Gamma_{\text{IEF}}$  at round  $t$ . Moreover, let us suppose that  $\gamma \in \Gamma_{\text{CF}}$ . That is,  $L(\gamma) > 0$ . More precisely, let  $i$ ,  $1 \leq i \leq \lfloor \log n \rfloor + 1$ , denote the smallest phase counter in the network, among all nodes. At phase  $i$ , there are at most  $n/2^{i-1}$  candidate leaders (i.e., at most this many roots). Thus,  $L(\gamma) = n/2^{i-1}$  at phase  $i$ . We have studied in the proof of Lemma 5 how Algorithm **CLE** performs the election process. After  $O(n \log n)$  rounds, Phase  $i + 1$  is completed, and the system reaches some configuration  $\gamma'$ . At this point, there remain at most  $n/2^i$  candidate leaders. Since  $L(\gamma') \leq n/2^i$ , we get that

$$L(\gamma') < L(\gamma).$$

The number of phases is upper bounded by  $\lfloor \log n \rfloor + 1$ . At phase  $\lfloor \log n \rfloor + 1$ , we reach a configuration  $\gamma''$  satisfying  $L(\gamma'') = 1$ . A direct consequence of Lemma 5 and Lemma 6 is that only the node  $\ell^*$  with maximum identity has  $\text{leader}_{\ell^*} = 1$ . Every other node  $v$  has  $\text{leader}_v = 0$ . Moreover, for every node  $v \neq \ell^*$ , we have  $\mathbf{p}_v \neq \emptyset$ , and the structure induced by the pointers  $\mathbf{p}_v$ , for all  $v \neq \ell^*$  forms a spanning tree rooted in  $\ell^*$ . Regarding time complexity, our algorithm takes  $O(n \log n)$  rounds to detect an impostor leader,  $O(n)$  rounds to reset the system after the detection of an error, and  $O(n \log^2 n)$  rounds to elect the leader. Therefore, in total, Algorithm **CLE** performs  $O(n \log^2 n)$  rounds to converge to the leader specification.  $\square$

**Lemma 8**  $\Gamma_{\text{LE}}$  is closed.

**Proof.** The rule  $\mathbb{R}_{\text{Passive}}(v)$  is the only rule performed by  $v$  that modifies the distance and the leader variables of node  $v$ . Let  $\ell^*$  be the node with the maximum identity. As a direct consequence of Lemma 7, in the initial configuration,  $\ell^*$  is the only node that has  $\mathbf{d}_{\ell^*} = 0$  and  $\text{leader}_{\ell^*} = 1$ . In other words  $\ell^*$  is the only elected node. Therefore,  $\ell^*$  changes the phase of the system by increasing the current phase, or by restarting from phase 1 (see predicate **T.Inc** and command **Inc**). Thus, every node  $v$  satisfies  $\text{Bit-Strong}_v \leq \text{Bit-Strong}_{\ell^*}$ . Moreover, for every phase  $i$ ,  $1 \leq i \leq \lfloor \log n \rfloor + 1$ , every node  $v$  satisfies  $\text{Bit}(i, \text{ld}_v) < \text{Bit-Position}_{\ell^*}$ . Hence, every node can only execute the command **Update**. Finally, nodes never change their distance, parent, and leader variables.  $\square$

## 4.1 Memory requirements

**Lemma 9** Algorithm **CLE** use  $O(\log \log n)$  bits of memory per node.

**Proof.** Algorithm **CLE** has two types of variables: the variables that use a constant number of bits, and those that use  $O(\log \log n)$  bits. Variables of the first type are:

$$\mathbf{p}_v \in \{\emptyset, 0, 1\}, \quad \mathbf{dB}_v \in \{0, 1\}, \quad \text{Add}_v \in \{+, ok, \emptyset\}, \quad \text{and} \quad \text{leader}_v \in \{0, 1\}.$$

Variables of the second type are:

$$\widehat{\mathbf{B}}_v \in \{1, \dots, \lfloor \log n \rfloor\}, \quad \mathbf{PL}_v \in \{1, \dots, \lfloor \log n \rfloor\} \times \{0, 1\}, \quad \mathbf{HC}_v \in \{1, \dots, \lfloor \log n \rfloor\} \times \{0, 1\},$$

and

$$\text{Elec}_v \in \{1, \dots, \lfloor \log n \rfloor\} \times \{1, \dots, \lfloor \log n \rfloor\} \times \{1, \dots, \lfloor \log n \rfloor\} \times \{0, 1\}.$$

Hence, **CLE** uses  $O(\log \log n)$  bits of memory per node.  $\square$

## 5 Conclusion

In this paper, we have shown that, in the state model, with a weakly fair distributed scheduler, one can elect a leader in a ring with a (non-silent) self-stabilizing algorithm using only  $O(\log \log n)$  bits of memory per node. It is known that one cannot do the same using only  $O(1)$  bits of memory per node (see [8]). An intriguing question is whether one can perform leader election in the same framework as in this paper, using just  $o(\log \log n)$  bits per node, and, if yes, to what extent can the memory requirement for (non silent) leader election being reduced. A natural candidate function for the minimum memory requirement for leader election is  $O(\log^* n)$  bits per node, by applying the techniques in this paper recursively. This however seems to be non trivial, as self-stabilization has to be maintained at every level of the recursion.

## References

- [1] J. Adamek, M. Nesterenko, and S. Tixeuil. Using abstract simulation for performance evaluation of stabilizing algorithms: The case of propagation of information with feedback. In *SSS 2012*, LNCS. Springer, 2012.
- [2] Y. Afek and A. Bremler-Barr. Self-stabilizing unidirectional network algorithms by power supply. *Chicago J. Theor. Comput. Sci.*, 1998.
- [3] A. Arora and M. G. Gouda. Distributed reset. *IEEE Trans. Computers*, 43(9):1026–1038, 1994.
- [4] M. Arumugam and S. S. Kulkarni. Prose: A programming tool for rapid prototyping of sensor networks. In *S-CUBE*, pages 158–173, 2009.
- [5] B. Awerbuch, S. Kutten, Y. Mansour, B. Patt-Shamir, and G. Varghese. A time-optimal self-stabilizing synchronizer using a phase clock. *IEEE Trans. Dependable Sec. Comput.*, 4(3):180–190, 2007.
- [6] B. Awerbuch and R. Ostrovsky. Memory-efficient and self-stabilizing network reset. In *PODC*, pages 254–263. ACM, 1994.
- [7] J. Beauquier, S. Delaët, S. Dolev, and S. Tixeuil. Transient fault detectors. *Distributed Computing*, 20(1):39–51, June 2007.
- [8] J. Beauquier, M. Gradinariu, and C. Johnen. Randomized self-stabilizing and space optimal leader election under arbitrary scheduler on rings. *Distributed Computing*, 20(1):75–93, January 2007.
- [9] L. Blin and S. Tixeuil. Brief announcement: deterministic self-stabilizing leader election with  $o(\log \log n)$ -bits. In *Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing, (PODC13)*, pages 125–127, 2013.
- [10] L. Blin and S. Tixeuil. Compact deterministic self-stabilizing leader election: The exponential advantage of being talkative. In *Proceedings of the 27th International Conference on Distributed Computing (DISC 2013)*, Lecture Notes in Computer Science (LNCS), pages 76–90. Springer Berlin / Heidelberg, 2013.
- [11] Y. Choi and M. G. Gouda. A state-based model of sensor protocols. *Theor. Comput. Sci.*, 458:61–75, 2012.

- [12] A. R. Dalton, W. P. McCartney, K. Ghosh Dastidar, J. O. Hallstrom, N. Sridhar, T. Herman, W. Leal, A. Arora, and M. G. Gouda. Desal alpha: An implementation of the dynamic embedded sensor-actuator language. In *ICCCN*, pages 541–547. IEEE, 2008.
- [13] A. Kumar Datta, L. L. Larmore, and P. Vemula. Self-stabilizing leader election in optimal space under an arbitrary scheduler. *TCS*, 412(40):5541–5561, 2011.
- [14] S. Devismes, T. Masuzawa, and S. Tixeuil. Communication efficiency in self-stabilizing silent protocols. In *ICDCS 2009*, pages 474–481. IEEE Press, 2009.
- [15] E. W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Commun. ACM*, 17(11):643–644, 1974.
- [16] S. Dolev. *Self-stabilization*. MIT Press, March 2000.
- [17] S. Dolev, M. G. Gouda, and M. Schneider. Memory requirements for silent stabilization. *Acta Inf.*, 36(6):447–462, 1999.
- [18] S. Dolev and T. Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago J. Theor. Comput. Sci.*, 1997, 1997.
- [19] S. Dolev, A. Israeli, and S. Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM J. Comput.*, 26(1):273–290, 1997.
- [20] S. Dubois and S. Tixeuil. A taxonomy of daemons in self-stabilization. Technical Report 1110.0334, ArXiv eprint, October 2011.
- [21] F. E. Fich and C. Johnen. A space optimal, deterministic, self-stabilizing, leader election algorithm for unidirectional rings. In *DISC*, pages 224–239. Springer, 2001.
- [22] M. G. Gouda, J. Arturo Cobb, and C. Huang. Fault masking in tri-redundant systems. In *SSS, LNCS*, pages 304–313. Springer, 2006.
- [23] T. Herman and S. V. Pemmaraju. Error-detecting codes and fault-containing self-stabilization. *Inf. Process. Lett.*, 73(1-2):41–46, 2000.
- [24] J. Hoepman. Self-stabilizing ring-orientation using constant space. *Inf. Comput.*, 144(1):18–39, 1998.
- [25] A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Inf. Comput.*, 104(2):175–196, 1993.
- [26] G. Itkis and L. A. Levin. Fast and lean self-stabilizing asynchronous protocols. In *FOCS*, pages 226–239. IEEE Computer Society, 1994.
- [27] G. Itkis, C. Lin, and J. Simon. Deterministic, constant space, self-stabilizing leader election on uniform rings. In *WDAG, LNCS*, pages 288–302. Springer, 1995.
- [28] Amos Korman, Shay Kutten, and Toshimitsu Masuzawa. Fast and compact self stabilizing verification, computation, and fault detection of an mst. In *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011*, pages 311–320, 2011.
- [29] T. Masuzawa and S. Tixeuil. On bootstrapping topology knowledge in anonymous networks. *ACM Transactions on Adaptive and Autonomous Systems*, 4(1), 2009.

- [30] A. J. Mayer, Y. Ofek, R. Ostrovsky, and M. Yung. Self-stabilizing symmetry breaking in constant-space (extended abstract). In *STOC*, pages 667–678, 1992.
- [31] T. M. McGuire and M. G. Gouda. *The Austin Protocol Compiler*, volume 13 of *Advances in Information Security*. Springer, 2005.
- [32] S. Tixeuil. *Algorithms and Theory of Computation Handbook*, pages 26.1–26.45. CRC Press, Taylor & Francis Group, 2009.